MASTER EN SCIENCES INFORMATIQUES

Moteur de recherche sémantique

OtJacques, Nicolas

Award date:
2009

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'informatique
Année académique 2008-2009

Moteur de recherche et sémantique

Nicolas Otjacques
<nicolas.otjacques@student.fundp.ac.be>

Mémoire présenté en vue de l'obtention du grade de Master en Science Informatique

Abstract (fr)

Le but de ce travail est de montrer le développement d'un moteur de recherche documentaire personnalisé basé sur l'utilisation d'une ontologie et de la synonymie dans un domaine spécifique et limité. Ainsi que de montrer son évaluation en comparaison de moteurs de recherche documentaire « classiques » existants.

Mots clés : indexation, ontologie, moteur de recherche (documentaire), sémantique

Abstract (en)

The goal of this work is to show the development of an search engine for textual document based on a ontology and the usage of synonyms. This engine is constructed for a specific and limited domain. More over, this work show an evaluation of the developed engine in a comparison between it and classical existing search engines.

Keywords: indexation, ontology, search engine, semantic

Remerciements

Je tiens dans un premier temps à remercier monsieur ***Hainaut*** *pour le suivi qu'il a fait tout au long de mon mémoire.*

De même, je remercie messieurs ***Estiévenart*** *et* ***Roumier*** *du* Centre d'Excellence en Technologie de l'Information et de la Communication (CETIC) *qui m'ont accueilli et guidé depuis le stage qui a servi de point de départ de ce travail.*

Table des matières

Table des figures

Liste des tableaux

Glossaire

A

API Application programming interface. 41, 42, 49, 52, 53

B

Bug Tracker Logiciel permettant de planifier et d'effectuer un suivi dans le cadre d'un travail collaboratif en y renseignant les nouvelles fonctionnalités à implémenter ou les bugs à corriger et en pouvant suivre l'évolution de ses « tâches ». 95

C

C Langage de programmation. 14, 28
CETIC Centre d'Excellence en Technologie de l'Information et de la Communication. 3, 15, 93–95
crawler outil parcourant le `Web` à la recherche de documents. Parfois appelé aussi spider. 28, 50

D

Data Property Construction propre à *Web Ontology Language (OWL)* correspondant à un attribut d'un concept ayant pour valeur un type simple (et donc pas une instance d'un concept). 55, 56, 60

E

Excel Tableur de la suite *MicroSoft (MS)* Office. 53

F

framework ensemble de librairies, modèles, mécanismes, conventions, ... déployées dans le but de faciliter ma réalisation d'une tâche particulière (ou d'un ensemble de tâches) en informatique. 14, 66, 67, 69
FUNDP Facultés Universitaires Notre-Dame de la Paix. 50, 95

H

HTML HyperText Markup Language. 53, 54

I

indexation Processus lié à la création d'un *index*. 19–22, 26–28, 42, 44, 47, 48, 83

index Structure technique ayant pour but d'accélérer l'accès aux éléments pour lesquels il est construit. Voir section 2.2 pour avoir une description complète. 13, 14, 19–22, 25–28, 35, 42–44, 50, 55, 69, 70, 72, 89

J

Java Langage de programmation orienté objet. 14, 15, 27, 28, 48, 52–55, 65, 66, 69
Jena *framework* permettant d'interroger une ontologie. 66, 67
JSP JaveServer Page. 69, *voir aussi* servlet

L

Lucene Libraire open-source développée par la *Fondation Apache* permettant de travailler avec un *index* respectant un format spécifique (et ouvert). Voir la section 2.2.3 pour plus de détails. 14, 15, 27, 28, 42, 44, 51, 55, 68, 69, 71, 89, 94
Lucy Portage *C* de la librairie *Lucene*. 28
Luke Outil développé en *Java* permettant de lire et même de modifier un *index* respectant le format de *Lucene*. Voir la section 2.2.3 pour plus de détails. 20, 28

M

mot de liaison Les mots de liaison sont propres à une langue. Ce sont des mots généralement courts qui permettent de faire le lien entre deux autres mots. Par exemple : « le, la, les, du, en, sûr, ... » en français.. 15, 25, 26
MS MicroSoft. 13, 15, 16, 53

O

Object Property Construction propre à *OWL*. Relation depuis un concept vers un autre concept (éventuellement le même). 60
ODT OpenDocument Text. 54
ontologie Représentation formelle d'un domaine de connaissance. 14, 15, 18, 19, 25, 29–34, 39–45, 47, 48, 55–58, 61, 64, 65, 67–73, 75, 76, 78, 79, 81–84
OWL Web Ontology Language. 13, 14, 34, 55, 60, 65
OWL-DL OWL Description Logic. 34, 40, 44
OWL-F OWL Full. 34
OWL-L OWL Light. 34

P

parser outil effectuant la tâche de *parsing*. 42, 52–54, 89
parsing analyse d'un conteneur (fichier par exemple) pour en extraire le contenu (texte par exemple). 14, 25, 52, 54, 89
pattern Patron, motif, structure, ... Si l'on prend le cas d'une *Expression Régulière (RegExp)*, ce pattern défini un ensemble de chaîne de caractères. 52
PDF Portable Document Format. 22, 42, 53, 54
Pellet *reasoner* *Open-Source* travaillant sur les *ontologies* de la famille *OWL*. 66, 67
PHP Langage de script libre principalement utilisé pour produire des pages web dynamiques via un serveur HTTP, mais pouvant également fonctionner comme n'importe quel

langage interprété de façon locale, en exécutant les programmes en ligne de commande. PHP est un langage impératif disposant depuis la version 5 de fonctionnalités de modèle objet complètes. En raison de la richesse de sa bibliothèque, on désigne parfois PHP comme une plate-forme plus qu'un simple langage. (extrait de *wikipedia*). 28, 69
POI Poor Obfuscation Implementation. 53, 54
Powerpoint Logiciel de présentation de la suite *MS* Office. 53, 54
Protégé outil permettant de créer et modifier des *ontologies*. 57, 61, 62
PyLucene Portage *Python* de la librairie *Lucene*. 28
Python Langage de programmation. 15, 28

R

RALI Laboratoire de Recherche Appliquée en Linguistique Informatique. 96
reasoner Outil permettant de constuire des raisonnements à partir de connaissances (par exemple une *ontologie*). 14, 66
RegExp Expression Régulière. 14, 23, 52, 69
RI Recherche d'information. 20, 21, 25, 34–36
RSS désigne une famille de formats – généralement basés sur *eXtensible Markup Language (XML)* – utilisés pour la syndication de contenu `Web`. Ce standard est habituellement utilisé pour obtenir les mises à jour d'information dont la nature change fréquemment. Pour les recevoir, l'utilisateur doit s'abonner au flux, ce qui lui permet de consulter rapidement les dernières mises à jour, à l'aide d'un `agrégateur`, sans avoir à se rendre sur le site. (extrait de *wikipedia*). 22, 49–51, 94
RTF Rich Text Format. 53, 54

S

SAX Simple API for XML. 53, 54
servlet Objets *Java* qui gèrent de manière dynamique des requêtes et des réponses sur un serveur `Web`. 69
SPARQL SPARQL Protcol and RDF Query Language. 67
SQL Structured query language. 67–69
stopword Expression anglaise pour *mot de liaison*. 25, 26, 54, 55
SVN Système de gestion de versions. 95

T

taxonomie Science, lois, ou principes de classification. Systématique. Division en groupes ordonnés ou en catégories. 29, 32, 40, 59, 62
TF-IDF Term Frequence - Inverse Document Frequency. 36, 71, 87
TIM Technical Information Meeting. Séminaire organisé par les employés (stagiaire inclus) du *CETIC* pour leur collègues dans un domaine dans lequel ils travaillent. 95
TSI Traitement sémantique de l'information. 93, *voir aussi* CETIC

U

UML Unified Modeling Language. 24, 29
URL Uniform Resource Locator. 40, 49, 50, 52, 55, 71, 76

W

W3C World Wide Web Consortium. 34
Word Traitement de texte de la suite *MS* Office. 22, 53, 54

X

XML eXtensible Markup Language. 15, 34, 53, 54

Chapitre 1

Introduction

Une personne marche dans la rue. Elle entend par hasard une conversation de deux personnes qu'elle croise dans la rue. Les propos éveillent sa curiosité. Elle aimerait bien en savoir plus mais elle continue son chemin. Arrivée chez elle, elle s'installe devant son ordinateur, se connecte sur `Internet` et ouvre son moteur de recherche documentaire favoris. Elle entre alors une question relative à ce qui a éveillé sa curiosité dans la zone de saisie. Elle soumet la requête et se rend compte qu'il n'y a que deux ou trois résultats et que ça ne répond pas du tout à sa question. Elle remplace alors la question par une série de mots-clés qu'elle pense devoir trouver dans les documents traitant du sujet qui l'intéresse et soumet de nouveau la requête. Cette fois-ci, elle obtient des millions de résultats. Elle commence à regarder les quelques premiers documents et se rend compte que ça ne correspond pas vraiment à ce qu'elle a en tête. Elle revient à la requête et modifie un peu la série de mots-clés pour essayer d'obtenir un meilleur résultat. Elle finira probablement par trouver la réponse à sa question après plusieurs affinement de sa requête et l'épluchage des premiers résultats de chaque « version » de sa requête.

Ce scénario met en avant un problème qui peut être rencontré par des utilisateurs lors de la recherche d'information sur `Internet`. Chaque fois que l'on se lance dans une recherche d'information pour trouver la réponse à une question, on doit franchir la distance qui sépare la question telle qu'on se la représente dans notre tête à une expression « compréhensible » par le système informatique en face de nous pour que celui-ci puisse trouver des documents dans lesquels il juge que l'on pourra trouver notre réponse. Comme dans le scénario précédent, il se peut très probablement que la première formulation que l'on soumettra au moteur de recherche documentaire ne retourne pas de résultats concluants pour nous et on aurait dès lors tout intérêt à tenter de soumettre notre requête au système informatique avec une autre formulation pour voir si celle-ci ne nous donnerait pas de « meilleurs » résultats.

La façon de formuler une requête dans un moteur de recherche a aussi une influence sur l'ordre des résultats. En effet, l'ordre dans lequel sont affichés les résultats a une grande importance. Les requêtes classiques soumises sur des moteurs de recherche documentaire grand public sur `Internet` retournent des milliers voir des millions de résultats. On ne peut humainement pas se permettre de regarder l'ensemble de ses résultats les uns après les autres. On se contente bien souvent des quelques premiers résultats.

Chaque jour il y a de plus en plus de documents disponibles que ce soit sur `Internet` ou en interne. Trouver les bons documents quand on en a besoin devient de plus en plus important et de plus en plus difficile. Comment faciliter cette tâche de recherche ? Comment rendre les moteurs de recherche documentaire plus efficaces ? Une piste d'amélioration repose sur l'affinement des résultats. Par exemple en réduisant la distance entre ce que les utilisateurs ont dans la tête et la façon dont ils doivent exprimer leur requête pour obtenir les bons résultats.

Une approche possible pour diminuer cette distance entre la question de l'utilisateur et la requête du moteur de recherche documentaire passe par une formalisation des connaissances à l'aide d'une *ontologie* qui permettrait au système de mieux « comprendre » la demande de l'utilisateur.

Pour simplifier cette tâche, on peut commencer par se limiter à un moteur de recherche documentaire pour un certain domaine. Par exemple le domaine d'une entreprise ou d'un site `Web` proposant de nombreux contenus sur un domaine bien ciblé.

Dans ce travail, on va élaborer un moteur de recherche documentaire basé sur une *ontologie* pour essayer de montrer que cet apport de connaissances entraîne une utilisation plus facile et plus efficace du système de recherche d'information. On va commencer avec le chapitre 2 par parler des concepts de base. Ensuite, au chapitre 3, on va aborder les méthodologies qui seront appliquées dans le développement de notre outil, les différents éléments sur lesquels il repose et l'évaluation du moteur de recherche créé. Ensuite vient le chapitre 4 qui décrit les outils développés en terminant par le moteur de recherche documentaire. Et enfin, on finira par une petite étude de cas au chapitre 5 pour évaluer ainsi notre outil.

Chapitre 2

État de l'art

2.1 Introduction

Le présent chapitre a pour but de mettre en avant les éléments importants pour la suite du travail et de les placer dans le contexte de l'état actuel des recherches dans le domaine. Trois sujets majeurs sont abordés dans ce chapitre :

index Nous commençons à la section 2.2 par l'explication du principe de l'*index*. Ensuite, nous passons à l'application de ces principes à l'*indexation* de documents et finalement nous jetons un oeil aux différentes librairies et aux différents outils qui existent pour travailler avec un *index*.

ontologie Le second sujet majeur, traité à la section 2.3, commence par l'explication du concept d'*ontologie*, des points communs et des différences avec le concept des bases de données pour passer ensuite à la façon de construire une telle *ontologie*.

Moteur de recherche Finalement, nous nous attaquons, à la section 2.4, à la partie consultation reposant sur les concepts précédents pour permettre à l'utilisateur d'en tirer profit. Pour cela, il faut commencer par parler de la notion de requête avant de parler de la notion de pertinence qui permet alors de parler de la notion d'évaluation du système lui-même.

2.2 Index et indexation

2.2.1 Généralités

> « Ce terme *index* désigne un mécanisme dont l'objectif est comparable à celui de **l'index d'un ouvrage** tel que celui-ci. Étant donné un mot significatif, l'index nous donne la liste des numéros de page où ce mot apparaît, sans qu'il soit nécessaire de parcourir l'intégralité de l'ouvrage pour retrouver ces pages. En outre, l'index est ordonné de manière telle qu'on puisse rapidement y repérer le mot recherché.
>
> Un index est une structure technique, en général invisible pour le programmeur, qui permet un accès sélectif et rapide aux enregistrements d'un fichier qui possèdent des valeurs déterminées de certains champs [...] »
>
> [Hainaut, 2005, p. 107 – 108]

Pour reprendre ce qui a été dit dans la citation ci-dessus, certains ouvrages contiennent un *index*. Ce dernier est une liste des termes jugés importants avec les numéros des pages où apparaissent chacuns de ces termes. L'idée sous-jacente est que le lecteur peut vouloir chercher les passages parlant d'un de ces termes. Sans l'*index*, le lecteur devrait parcourir tout l'ouvrage page par page pour trouver chaque passage intéressant. Avec l'*index* qui lui indique les numéros de pages, il n'a plus qu'à parcourir quelques pages pour trouver ce qu'il cherche. Il y a un gain énorme de temps pour le lecteur. En reprenant la définition d'un *index* de la citation ci-dessus, une autre notion importante apparaît. Il faut que « l'*index* » soit « ordonné de manière telle qu'on puisse rapidement y repérer le mot recherché. » ce qui signifie pour un *index* « papier » que les termes qui le composent soient triés pour que le lecteur puisse trouver le terme qu'il cherche sans devoir parcourir tout l'*index* terme par terme. Un moyen simple et courant de trier l'*index* est d'organiser les termes par ordre alphabétique.

La technique de l'*index* a été portée dans le domaine de l'informatique. Notamment dans les domaines des bases de données et des moteurs de recherche documentaire (aussi appelé *Recherche d'information (RI)*). L'*index* n'est alors souvent plus visible de l'utilisateur mais le principe reste le même. L'utilisateur cherche un terme présent dans un document ou la valeur d'un champ présent dans un enregistrement d'une table d'une base de donnée. C'est le système qui recherche pour lui les documents ou les enregistrements qu'il possède correspondant à ce que l'utilisateur désire. S'il n'y a pas d'*index*, le système doit parcourir chaque document ou chaque enregistrement un par un jusqu'à trouver une réponse valable pour l'utilisateur. Avec un *index* par contre, il suffit au système de regarder dans celui-ci les éléments attachés à ce que l'utilisateur désire et de renvoyer directement ces éléments. La figure 2.1 illustre un *index* permettant de travailler sur un fichier séquentiel. Il est extrait de [Hainaut, 2005, page 109]. La figure 2.2 est une capture d'écran de l'outil *Luke* qui est un outil utilisant l'index d'un corpus de documents. La capture illustre la recherche du terme « management » dans un *index* précédemment généré. On peut observer un sous-ensemble des documents[1] de la collection où le terme apparaît.

Le processus visant à créer une structure d'*index* est appelée *indexation*. Elle vise à associer à chaque élément que l'on souhaite indexer (champ d'un enregistrement d'une base de données, mot contenu dans un document, ...) l'ensemble des ressources (enregistrement, document, ...) dans lesquelles l'élément est présent. Cette association est représentée par l'équation 2.1 ci-dessous.

$$\forall i : élément_i \rightarrow \{ressource \mid élément_i \in ressource\} \tag{2.1}$$

L'existence d'un *index* augmente donc considérablement la vitesse de recherche d'un élément présent dans celui-ci mais cela a un coût. La structure d'*index* consomme de l'espace de stockage supplémentaire à côté de l'information déjà stockée. D'après [Cao, 2004, page 2], cet espace supplémentaire varie entre 40 et 200% de la taille de l'ensemble indexé. L'auteur met cependant en avant le fait que l'espace de stockage coûte de moins

[1] Le sous-ensemble correspond au 19 documents jugés les plus « pertinents »

en moins cher et que cet inconvénient n'en est donc plus vraiment un aujourd'hui. Malheureusement, ce n'est pas le seul inconvénient lié à l'utilisation d'un *index*. Vu que la structure créée reflète directement la collection de ressources indexées, chaque fois qu'on modifie d'une manière où d'une autre la collection, il faut s'assurer que l'*index* reste valide. Si on ajoute une nouvelle ressource, il faut effectuer le processus d'*indexation* pour celle-ci et fusionner le résultat avec l'index existant. Si on enlève une ressource existante, il faut enlever de l'*index* toutes les références à cette ressources dans les ensembles associés aux éléments et vérifier qu'on a pas dans l'*index* d'éléments associés à un ensemble vide auxquels cas on doit également enlever l'élément de l'index. Enfin, si on modifie une ressource, on doit répercuter les changements sur l'*index*[2] de la même façon.

2.2.2 Indexation de document de type texte

L'*indexation* de documents consiste à créer un *index* relatif à un corpus de documents. Cette démarche s'inscrit clairement dans le cadre d'un système de *RI*. Comme précisé dans [Cao, 2004], ces documents peuvent être de natures variées : texte brut, page `Web`, image, vidéo, ...

On distingue d'un côté les documents « textuels » qui sont essentiellement composés de mots et de l'autre côté les documents « multimédia » qui contiennent des éléments plus

[2]par exemple en considérant qu'on a enlever la ressource « ancienne version » et qu'on a dans un même temps ajouté la ressource « nouvelle version ».

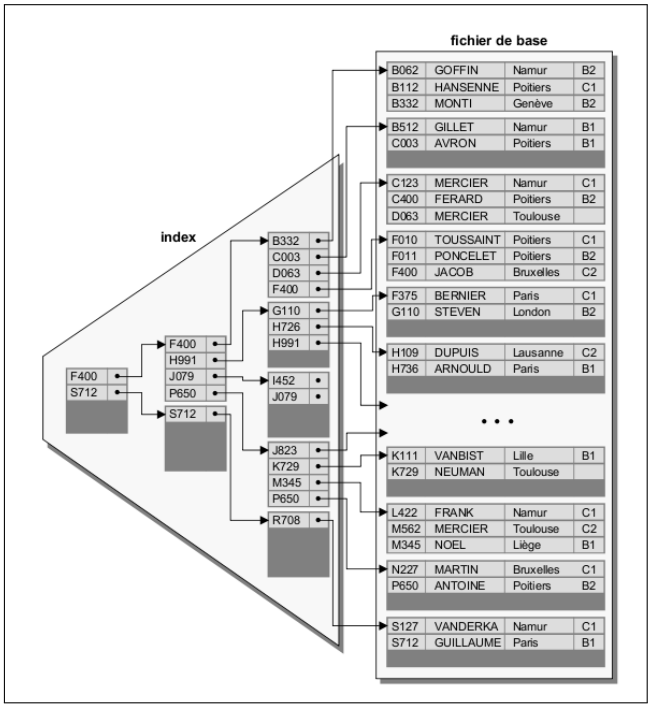

FIGURE 2.1 – Fichier séquentiel indexé extrait de [Hainaut, 2005]

élaborés (image, son, vidéo, ...). Dans ce travail nous nous focalisons sur les documents « textuels » pour lesquels les techniques d'indexation sont plus faciles à maîtriser.

Si nous nous concentrons sur les documents textuels, qu'ils soient structurés ou pas, nous pouvons enrichir la notion d'*indexation* en détaillant une série de traitements qui doivent être réalisés avant d'écrire (ou modifier) l'*index* à proprement parler.

La collecte des documents

La première chose à faire quand on veut indexer un corpus de documents c'est de définir le corpus. Le cas simple consiste à avoir l'ensemble du corpus directement disponible (par exemple dans un répertoire de son ordinateur) et de tout prendre. Mais avec les possibilités d'internet, les corpus de nos jours sont plus hétéroclites. On peut construire son propre corpus sur mesure en prenant les documents à droite et à gauche. On peut par exemple prendre quelques pages `Web` sur un site, des documents *Portable Document Format (PDF)* stockés sur un autre, les entrées d'un flux *RSS*, des documents réalisés avec *Word* sur la machine d'un collègue via un intranet, le contenu de ses propres mails, ... Une petite illustration est disponible à la figure 2.3.

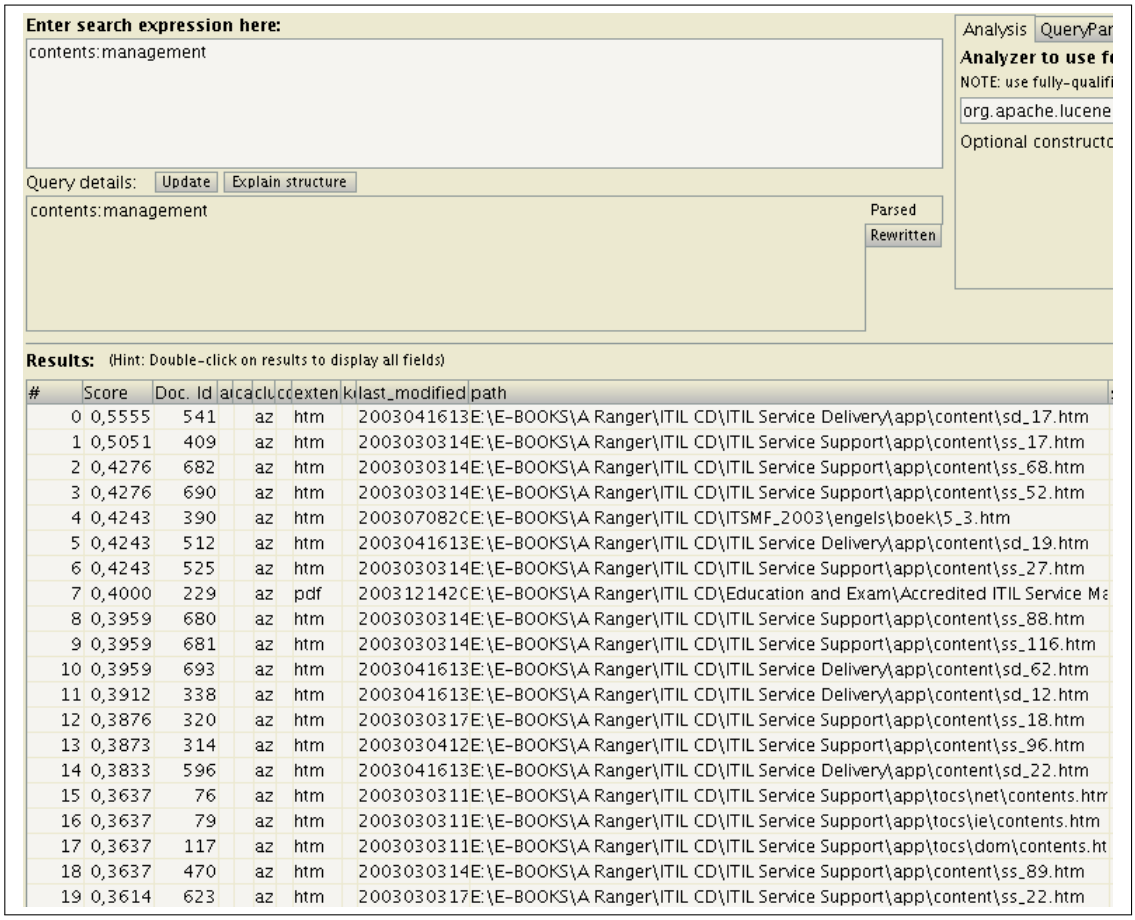

FIGURE 2.2 – Capture d'écran de l'outil Luke montrant la recherche du terme « management » dans un index généré à l'aide de Lucene

Nous remarquons que les documents auxquels nous pouvons accéder sont accessibles via un certain nombre de « chemins » différents. Et il faudra donc mettre au point un mécanisme de collecte de documents pour pouvoir utiliser chacun de ces chemins.

Une problématique importante ne peut être évitée quand on commence à parcourir le `Web` à la recherche de documents pour construire un corpus. Si on ne prend pas garde et qu'on s'arrange pour collecter tous les documents disponibles, on va se retrouver avec un processus qui va parcourir l'ensemble de l'internet pour récupérer tous les documents possibles. Cela va naturellement prendre un temps très conséquent et surtout ramener bien plus de documents inadéquats pour le corpus que de documents potentiellement pertinents. Il est donc important de se pencher sur le problème d'arrêt pour tous les systèmes de collecte de documents.

Le filtrage des documents

Avec l'essor des technologies et surtout d'internet, nous avons accès à un nombre très important de documents. Nous venons de voir que nous pouvions (et même que nous devions) limiter la taille du corpus en fixant un critère d'arrêt aux différents collecteurs utilisés. Mais récupérer « tous les documents qui se trouve à tel endroit » n'est pas toujours suffisant, parfois il faut un critère plus précis pour restreindre le corpus aux documents qui pourraient être utiles. Il peut donc s'avérer nécessaire de mettre en place des filtres qui épurent le corpus de documents de tous les éléments qu'on ne juge pas utile de garder. Par exemple, tous les documents qui ont une taille inférieure à une certaine valeur ou tous ceux qui respectent une certaine convention de nommage qu'on pourrait détecter à l'aide d'une *RegExp*, ...

On peut bien entendu cumuler les filtres définis pour avoir un résultat plus fin. Pour effectuer ce cumul, il y a deux méthodes principales.

La première est basée sur le principe des poupées russes : chaque filtre est contenu dans un autre filtre jusqu'à ce qu'on arrive au filtre « extérieur ». À partir de là, on se contente de communiquer avec le filtre extérieur qui va alors communiquer avec le filtre qu'il encapsule et ainsi de suite jusqu'au filtre le plus à l'intérieur. Ce filtre va parcourir la liste des documents jusqu'à en trouver un qui respecte son critère et le renvoyer au filtre dans lequel il est contenu. Si ce document respecte également le critère de ce filtre

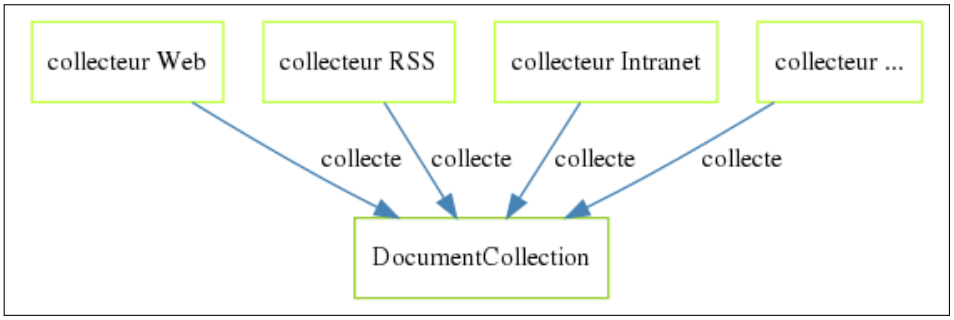

FIGURE 2.3 – Illustration du processus de collecte

là, le document continue sa route vers l'extérieur de la même manière, si par contre à un moment ou un autre le document ne respecte pas le critère d'un filtre, on refuse le document et on cherche un autre document. Cette démarche est illustrée par le diagramme de séquence *Unified Modeling Language (UML)* de la figure 2.4.

La seconde méthode consiste plutôt à mettre tous les filtres les uns à côtés des autres. On prend un document dans la liste et on regarde filtre par filtre s'il respecte le critère. S'il les respecte tous, le document est accepté et est incorporé dans le corpus. Dès qu'il ne respecte pas le critère d'un filtre, on « jette » le document et on recommence avec le suivant. Cette démarche est illustrée par le diagramme de séquence *UML* de la figure 2.5.

Choisir l'une ou l'autre de ces deux approches, ne change pas grand chose en terme de performance mais d'une personne à l'autre, l'approche la plus « naturelle » n'est pas la même.

Quand on choisit de cumuler plusieurs filtres pour avoir un résultat plus précis, il faut envisager l'ordre dans lequel les filtres vont être placés. L'ordre n'a pas d'influence sur le résultat car si un document est accepté c'est qu'il respecte les critères de l'ensemble des filtres sans exception et donc vérifier le critère d'un filtre *A* avant celui d'un filtre *B* ou l'inverse ne changera pas le fait que le document soit accepté ou non. En revanche, si

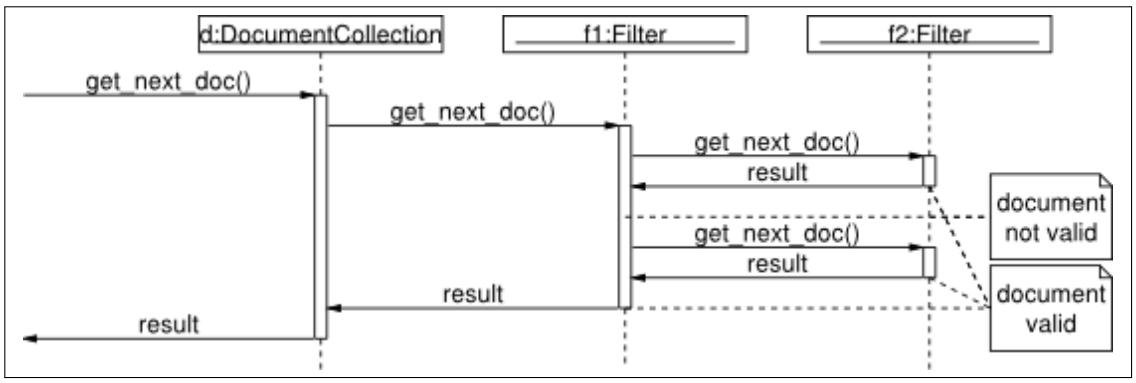

FIGURE 2.4 – Diagramme de séquence UML de l'approche « poupée russe » du cumul des filtres

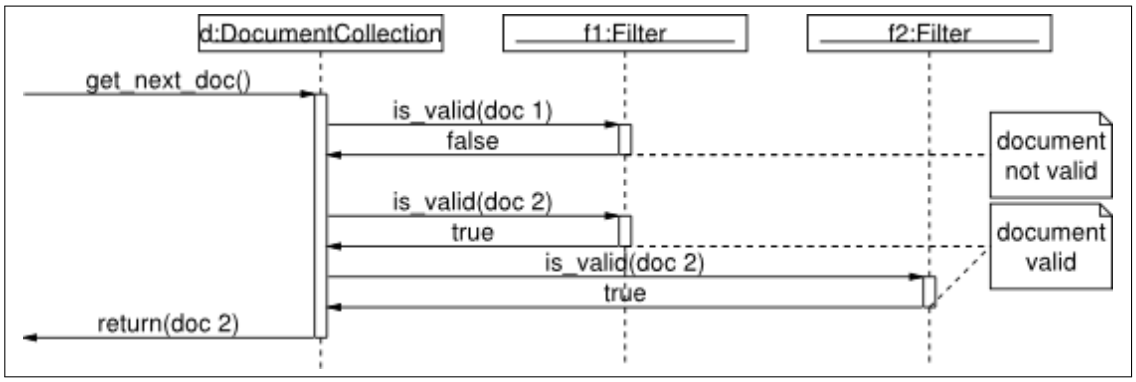

FIGURE 2.5 – Diagramme de séquence UML de l'approche en liste du cumul des filtres

l'évaluation du respect du critère du filtre *B* par un document nécessite énormément de temps et/ou de ressources et que l'évaluation correspondante pour le filtre *A* est immédiate et ne demande pas de grosses ressources, le fait de vérifier *A* avant *B* permet de ne pas perdre de temps ni de ressources pour faire les calculs pour le critère de *B* pour des documents qui ne respectent de toute façon pas le critère de *A*. On voit donc qu'on a tout intérêt à garder les filtres avec un traitement lourd pour la fin pour essayer d'éliminer le plus de documents possibles avant d'arriver à eux.

L'analyse des documents

Une fois qu'on a construit le corpus avec les documents que l'on souhaite, il y a encore une chose à faire avant de créer l'*index*. On ne peut pas se contenter de considérer que notre corpus est composé de documents textuels. Il y a plusieurs différences qui doivent être abordées pour avoir une analyse correcte.

Une première différence réside dans le format des documents. Il existe de très nombreux formats de documents textuels avec chacuns leur propre structure. Il n'existe pas de méthode magique pour extraire le contenu de tous ces différents formats. En reprenant [Cao, 2004], on peut avancer que la majorité des formats de documents textuels possède une première partie très structurée pouvant contenir des informations comme le titre, l'auteur, une liste de mots clés, ... Et à côté de ça, il y a une seconde partie généralement peu ou pas structurée qui est le contenu du document à proprement parler.

Toujours selon [Cao, 2004], le coeur de la *RI* repose sur la partie « contenu » qui est donc peu ou pas structurée. Mais la partie structurée est incluse dans la majorité des systèmes également.

Avant de pouvoir créer l'*index*, il faut donc mettre en place des mécanismes pour récupérer d'une part le contenu et d'autre part les données structurées associées[3] et ce pour chaque format de documents que l'on veut traiter. Heureusement, on trouve très facilement des librairies qui permette de lire sans trop de difficulté les différents formats utilisés actuellement. Le fonctionnement de la phase de *parsing* mêlée à celle d'analyse est représentée à la figure 2.6.

Une deuxième différence réside dans la langue des documents. On peut rencontrer dans un corpus des documents rédigés dans différentes langues. Pour une utilisation basique d'un *index*, on peut faire abstraction de la langue du document et indexer indifféremment des documents en français, en anglais et en néerlandais. Mais si on prend, par exemple, la peine d'enlever tous les *mots de liaison* – aussi appelés *stopwords* – qui n'apportent pas grand chose dans l'*index* pour l'alléger, on va directement faire face au problème de la langue car les *stopwords* changent d'une langue à l'autre.

Et au delà de ça, si on souhaite enrichir le système, par exemple à l'aide d'une *ontologie*, il deviendra aussi très important d'identifier la langue pour, par exemple, ajouter une phase de traduction vers le langage de l'*ontologie*.

Si l'on ajoute un traitement propre à la langue au processus d'analyse que nous avons déjà, la figure 2.6 devient la figure 2.7.

[3] qu'on appelle parfois méta-données

Une méthode pour détecter la langue est de se baser sur les *stopwords*. Ils sont bien connus et répertoriés sur le `Web`. Ils changent d'une langue à l'autre. Si bien qu'avec une simple analyse probabiliste des différents *mots de liaison* présents, on peut identifier la langue pour peux qu'on possède sa liste *stopwords*.

L'écriture de l'index

Arrivé ici dans le processus d'*indexation*, nous avons en main une collection d'informations pour chaque document. Il ne reste plus qu'à écrire tout ça dans un *index*. On peut bien entendu gérer soi-même la génération de cette structure mais le plus simple est d'utiliser une librairie qui nous donnera un résultat bien plus optimisé.

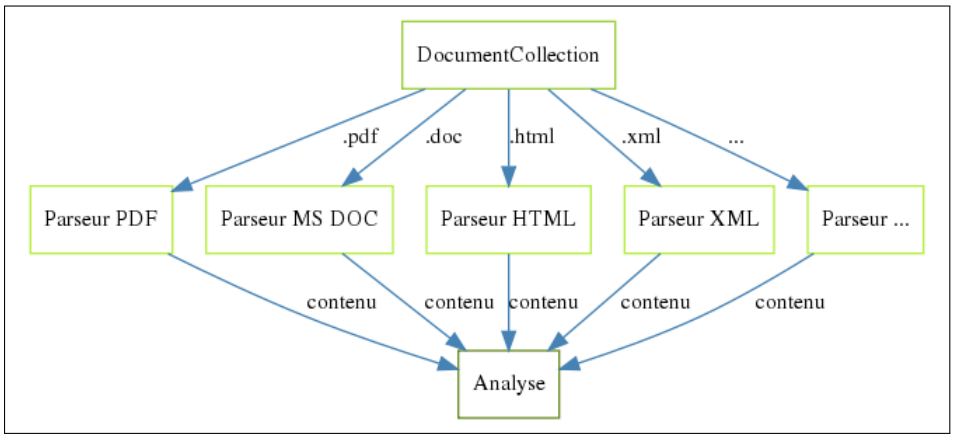

FIGURE 2.6 – Illustration du processus de parsing

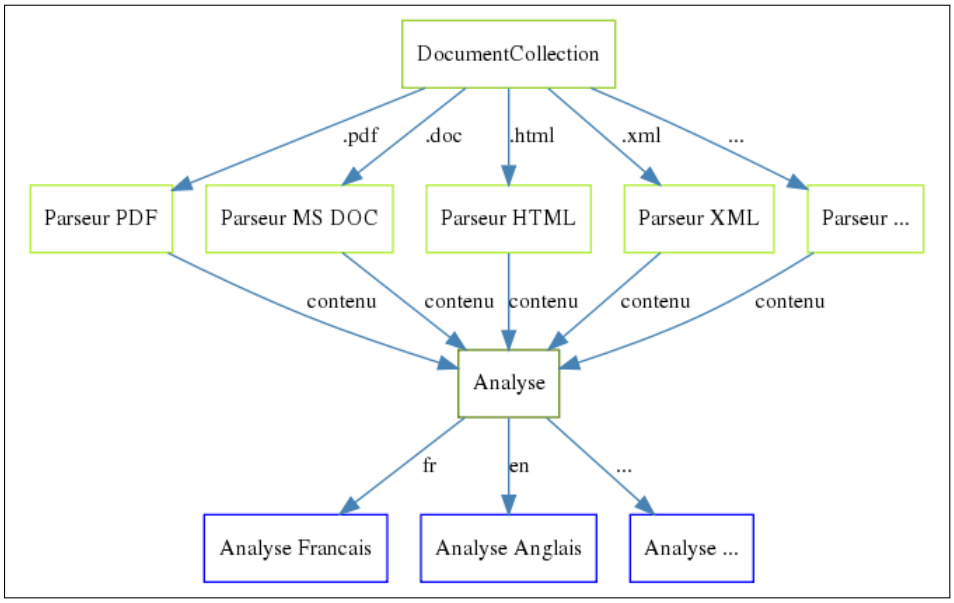

FIGURE 2.7 – Illustration du processus de parsing suivi d'une analyse propre à la langue

Résumé du processus d'indexation

Si on reprend le processus d'*indexation* depuis le début, on commence par collecter des documents comme illustré à la figure 2.3 après on filtre éventuellement la liste des documents collectés précédemment. On « parse » ensuite les documents en fonction du format comme illustré à la figure 2.6. Après cela on analyse en fonction de la langue comme illustré à la figure 2.7 et on finit le processus en écrivant le résultat dans un *index*. Ce qui nous donne au final la figure 2.8.

2.2.3 Librairies et outils liés à l'indexation

Lucene

Lucene est à la fois un projet open-source à l'initiative de la `Fondation Apache` et le logiciel phare de ce projet.

Le logiciel *Lucene* est une librairie pour le langage *Java* permettant l'*indexation* et la recherche dans l'*index* avec des fonctionnalités assez avancées comme la présentation d'extrait des documents.

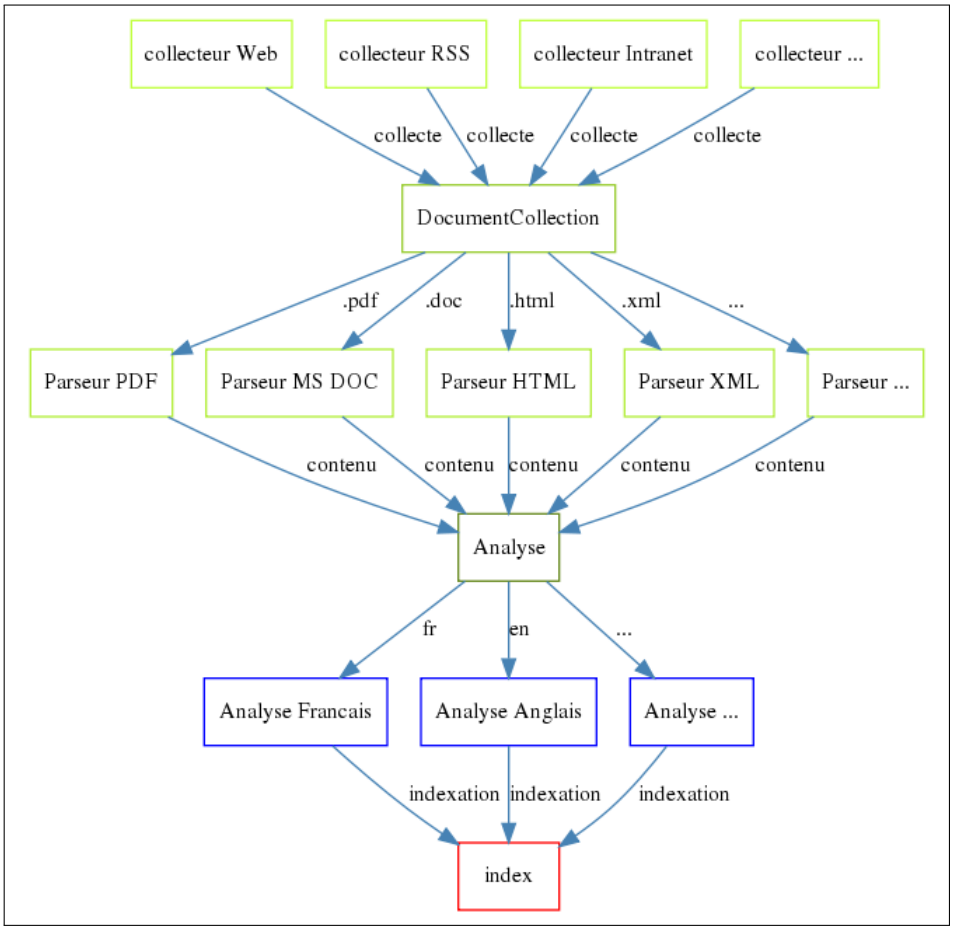

FIGURE 2.8 – Illustration du processus d'indexation

Le projet comporte une série de logiciels allant du *crawler* à la librairie d'*indexation* et même des moteurs de recherches complets. Et des portages du logiciel *Lucene* dans différents autres langages comme *Python* avec *PyLucene* ou *C* avec *Lucy*.

Il est à noter que le terme « Lucene » désigne aussi le format de l'*index* utilisé par l'ensemble des logiciels du projet. Ce projet est ouvert. Ce qui veut dire qu'on a beaucoup de libertés sur la façon de travailler avec : on peut, par exemple, le créer avec *PyLucene* puis le lire et le modifier avec *Lucy* et enfin créer son propre outil vu que la spécification du format est ouverte.

Le format d'*index Lucene* est utilisé couramment, notamment par :

- Le `framework Zend`[4].
- Le logiciel `Eclipse` pour parcourir sa documentation.
- `Wikipedia`

Pour plus d'informations sur *Lucene*, consulter [Hatcher et Gospodnetić, 2004, luc, 2009].

Luke

Parmi les autres outils mis à disposition dans le cadre du projet *Lucene*, on peut citer *Luke*. Il s'agit d'un utilitaire réalisé en *Java* et reposant entièrement sur la librairie *Lucene*. *Luke* fonctionne seul et permet de parcourir et modifier de manière conviviale un *index* qui respecte le format *Lucene*. Une capture d'écran de l'interface de cet outil se trouve à la figure 2.2.

Cet outil est particulièrement pratique lors des phases de développement et de diagnostique d'un outil utilisant un tel *index*.

2.3 Les ontologies

2.3.1 Généralités

> « [...] une **ontologie** est une description formelle explicite des concepts dans un domaine du discours (**classes** (appelées parfois **concepts**)), des propriétés de chaque concept décrivant des caractéristiques et attributs du concept (**attributs** (appelés parfois **rôles** ou **propriétés**)) et des restrictions sur les attributs (**facettes** (appelées parfois **restrictions de rôles**)). Une ontologie ainsi que l'ensemble des **instances** individuelles des classes constituent une **base de connaissances**. Il y a en réalité une frontière subtile qui marque la fin d'une ontologie et le début d'une base de connaissances. »
>
> [Noy *et al.*, 2005, p. 3]

[4] pour le *PHP*

La première chose à dire quand on aborde le sujet des *ontologies*, c'est qu'il existe une assez grande variété de définitions différentes associées à ce concept et que certaines d'entre elles se contredisent. Pour ce travail, nous allons reprendre la définition ci-dessus extraite de [Noy *et al.*, 2005]. Cette définition est similaire à celle de [Cimiano, 2006, chapitre 2]. On peut également citer la définition de [Gruber, 1995] qui définit une *ontologie* d'une manière plus pragmatique de la manière suivante : « une *ontologie* définit le vocabulaire avec lequel les requêtes et les assertions peuvent être échangées entre des agents. L'*ontologie* peut dès lors être vue comme un accord entre les agents pour utiliser un vocabulaire commun d'une manière cohérente et consistante. »

Si l'on se concentre sur la définition de [Noy *et al.*, 2005], on y voit un certain nombre de notions qui sont pour la plus part assez proches de ce que l'on peut trouver en *UML* ou d'autres modélisations du même genre. Par exemple les classes qui sont organisées en une *taxonomie* comme illustré à la figure 2.9.

La distinction entre facette et attribut mérite d'être précisée. En modélisation classique comme *UML*, on décrit un attribut en lui donnant un nom, un type, une cardinalité, ... pour une *ontologie*, on a d'un côté l'attribut et de l'autre un ensemble de facettes qui décrivent plus précisément l'attribut. Quelques exemples de facettes fréquentes d'un attribut sont la cardinalité, le type de valeur autorisé, le domaine, ...

En fait, la distinction entre facette et attribut permet d'avoir une portée plus universelle de l'attribut. Prenons l'exemple du domaine. On a une classe *A* pour laquelle on définit un attribut pour le nom. Plus tard on définit une classe *B* à un endroit très éloigné de la *A* mais on a besoin également de caractériser le nom de cette classe. On ne défini pas un deuxième attribut qui représentera le nom de cette classe mais on ajoute cette classe

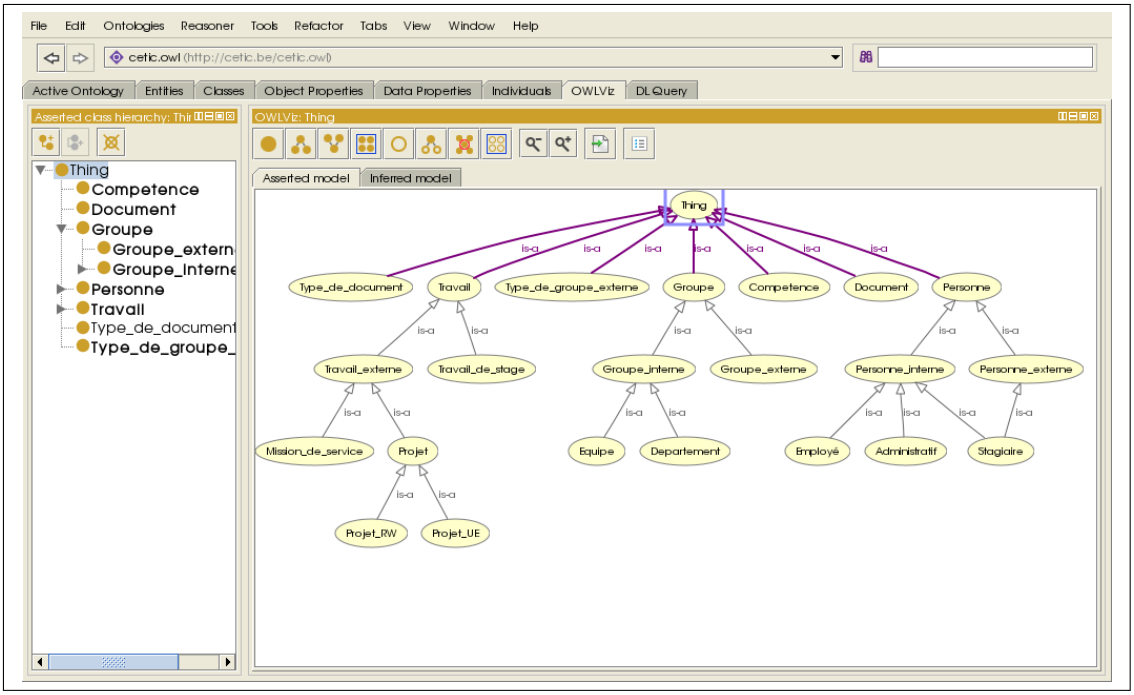

FIGURE 2.9 – Taxonomie de classes dans une ontologie

au domaine (formellement à la facette caractérisant le domaine) de l'attribut qui sera dès lors présent dans les deux classes. Cela est illustré à la figure 2.10.

Attaquons nous maintenant à une rapide comparaison des *ontologies* par rapport aux bases de données. L'idée de base est sensiblement la même, avoir une structure permettant de stocker ce qu'on sait sur un domaine. Les notions de classes, attributs (et facettes) et de relations sont présentes des deux côtés et remplissent des rôles assez similaire.

La principale différence entre les deux approches réside selon [Spyns *et al.*, 2002] dans la possibilité de faire de l'inférence dans les *ontologies*.

Attaquons-nous à l'inférence et prenons l'exemple suivant : on doit modéliser un travail effectué pour une certaine entreprise avec la collaboration d'une personne de cette entreprise. Il parait normal de spécifier la contrainte que le collaborateur en question doit bien appartenir à l'entreprise pour laquelle le travail est effectué. Supposons maintenant que ce collaborateur travaille dans plusieurs entreprises. Pour pouvoir exprimer la contrainte dans une base de données classique, il faudrait mettre en place des entités spécifiques pour cela. Dans une *ontologie*, il suffit simplement de restreindre le domaine ou l'image d'une relation ou d'un attribut pour qu'il soit lié non pas à l'ensemble des instances des collaborateurs possibles mais uniquement à un sous-ensemble qui représente ceux qui travaillent effectivement pour l'entreprise en question. C'est comme si on avait créé implicitement une sous-classe de la classe des collaborateurs possibles qui serrait la classe des collaborateurs possibles travaillant dans l'entreprise en question.

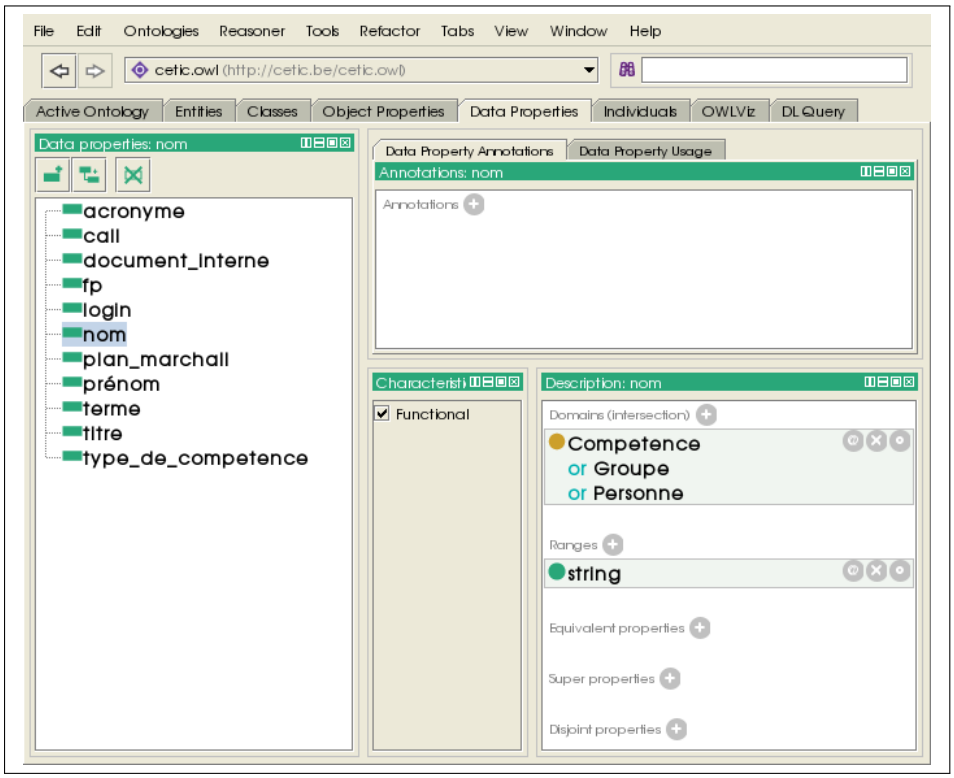

FIGURE 2.10 – Partage d'un même attributs entre deux classes

2.3.2 Construction d'une ontologie

Le but d'une *ontologie* étant d'être aussi largement utilisée que possible, la première chose à faire avant de se lancer dans la construction d'une *ontologie* est de rechercher s'il n'existe pas déjà une *ontologie* qui conviendrait à ce qu'on veut faire même en partie. Auquel cas on peut reprendre cette *ontologie* totalement ou en partie de même qu'on peut rajouter des éléments sur cette base.

Après avoir fait la première étape de recherche d'une *ontologie* préexistante, il se peut que l'on en arrive à construire sa propre *ontologie*. Ce qu'on s'apprête à construire est un modèle de la réalité et on entre dès lors dans un processus de construction itératif et rempli de choix (implicites et explicites) guidés par un objectif. Il faut en être bien conscient si on veut pouvoir faire les bons choix et avoir une *ontologie* aussi réutilisable que possible.

Méthodologie de construction proposée par [Noy *et al.*, 2005]

Cette méthodologie de construction est celle décrite dans l'article [Noy *et al.*, 2005]. La première remarque à faire sur cette méthodologie est qu'elle ne s'annonce pas comme la méthode ultime mais bien comme une parmi d'autre. L'accent est mis sur trois principes de base à garder à l'esprit tout au long de la construction de l'*ontologie* et même par la suite :

- « Il n'y a pas qu'une seule façon correcte pour modéliser un domaine - il y a toujours des alternatives viables. La meilleure solution dépend presque toujours de l'application que vous voulez mettre en place et des évolutions que vous anticipez. »
- « Le développement d'une *ontologie* est nécessairement un processus itératif. »
- « Les concepts dans une *ontologie* doivent être très proches des objets (physiques ou logiques) et des relations dans votre domaine d'intérêt. Fort probablement ils sont des noms (objets) ou verbes (relations) dans des phrases qui décrivent votre domaine. »

Une fois ces trois principes bien en tête, on peut se lancer dans la construction de l'*ontologie* en sept étapes.

Étape 1. Pour commencer il faut se poser une série de questions de base pour essayer de cibler le domaine et la portée de l'*ontologie*. Il est encore précisé que les réponses à toutes ces questions ne sont pas figées une fois pour toute mais bien qu'elles peuvent varier au cours de la construction.

Parmi les questions pointées dans l'article, une semble particulièrement intéressante : « À quels types de questions l'*ontologie* devra-t-elle fournir des réponses ? ». La réponse à cette question (qui est en fait une liste de questions) est très importante. Une fois l'*ontologie* terminée, on pourra utiliser cette liste pour tester l'*ontologie*.

Étape 2. Comme nous l'avons dit dans l'introduction de cette section, il est important d'envisager de réutiliser une *ontologie* existante avant de se lancer dans un lourd travail de construction. L'étape 2 de cette méthodologie consiste à chercher après une ou plusieurs *ontologies* qui correspondraient au domaine et à la portée spécifiée à l'étape 1 et d'essayer de repartir de celles-ci.

Étape 3. Cette étape consiste à lister tous les termes importants de l'*ontologie* qu'on est en train de construire et de se mettre d'accord sur les propriétés et la signification de chacun de ces termes. À ce niveau, on ne doit pas se préoccuper de savoir si les termes identifiés sont des classes ou des attributs pas plus que d'éventuel chevauchement entre les concepts. On se contente juste d'énumérer tous les termes importants de la manière la plus exhaustive possible.

Étape 4. Maintenant, on va définir les classes de l'*ontologie* et les organiser entre-elles de manière hiérarchique. Pour définir les classes, on se reporte à la liste de l'étape 3 et on y sélectionne « les termes qui décrivent des objets ayant une existence indépendante plutôt que les termes qui décrivent ces objets ».

Une fois les classes définies, il faut les organiser en une *taxonomie*. Cela peut se faire par une approche descendante, en commençant donc par mettre en haut les classes les plus générales puis en raffinant petit à petit. On peut aussi adopter une approche ascendante qui part des classes les plus spécifiques et les regrouper petit à petit pour revenir en haut avec les classes les plus générales. Et enfin on peut prendre une approche mixte qui se concentre sur les concepts les plus marquants et on rayonne autour de chacuns d'eux vers le haut ou vers le bas suivant les cas.

D'après l'article, il n'y a pas de méthode meilleure ou moins bonne qu'une autre dans les trois qu'on vient de citer. C'est au concepteur de choisir la méthode qui lui convient le mieux en fonction de la façon dont il perçoit lui-même le domaine.

Il y a quelques recommandations supplémentaires concernant la création de la hiérarchie de classes. Se fixer une convention de nommage pour les classes comme par exemple toujours utiliser le singulier. Une autre recommandation est d'éviter les boucles dans la hiérarchie. Et enfin, en général[5] le nombre de sous-classe acceptable pour une classe est entre deux et une douzaine. S'il n'y a qu'une seule sous-classe, ça ne vaut peut-être pas la peine de la préciser et s'il y en a trop, ajouter des niveaux intermédiaires peut s'avérer utile voir nécessaire.

En plus des ces remarques, l'article tient à rappeler que dans le domaine des *ontologies*, l'héritage multiple est autorisé et qu'il faut donc considérer ce cas. De même qu'une instance peut appartenir à plusieurs classes en même temps et il faut donc en tenir compte également particulièrement en définissant les cumuls interdits. Par exemple dire qu'une instance de vin rouge ne peut pas être en même temps une instance de vin blanc et inversement.

Étape 5. Une fois les classes définies et organisées de manière hiérarchique, on peut affiner les classes en définissant leurs propriétés. Pour ce faire, on va retourner à la liste de termes de l'étape 3 dans laquelle on a déjà puisé pour définir les classes. La plupart des

[5] Il y a donc bien entendu des exceptions

termes restants sont probablement les propriétés que l'on cherche. Pour chaque terme que l'on identifie comme étant une propriété, il reste à trouver dans quelle(s) classe(s) il faut la mettre. Une fois la liste épurée et en regardant les classes ainsi enrichies de propriétés, on peut découvrir des propriétés en plus auxquelles on n'avait pas pensé en rédigeant la liste de l'étape 3.

Il est à noter qu'une propriété doit être placée dans la classe la plus générale de la liste des classes pouvant avoir cette propriété. Elle sera dès lors naturellement héritée par toutes les classes « filles ».

Étape 6. Une fois les attributs définis et attachés à des classes, il reste à caractériser ces attributs à l'aide de facettes comme le type, le domaine[6], l'image, ...

Il est à noter que si les attributs sont hérités d'une super-classe à une de ses sous-classes, les facettes de cet attribut peuvent être plus restrictives au niveau des sous-classes. Par exemple, un attribut aurait une cardinalité minimale de 0 et maximale de 10 au niveau de la super-classe et on ferait 2 sous-classes : une première ayant une cardinalité maximale de 0 pour cet attribut, et une autre ayant une valeur comprise entre 1 et 10. Un autre exemple serait de dire que les valeurs autorisés pour un attribut sont des instances de $A \cup B$[7] et de définir une sous-classe pour laquelle toutes les valeurs de l'attribut en question appartiennent à $A \setminus B$[8].

Étape 7. On a maintenant la structure complète de l'*ontologie*, il est donc temps d'y insérer des instances en référençant bien la ou les classes à laquelle/auxquelles l'instance appartient et l'ensemble des valeurs des différents attributs.

Quand créer une sous-classe ?

Un point de vue complémentaire sur la création de la hiérarchie de classe expliquée à l'étape 4 de la section précédente est décrit dans [Bachimont, 2000]. L'accent porte sur les points communs et les différences entre un concept[9] et ses sous-concepts et entre les différents sous-concepts d'un même concept.

« La communauté avec le père » : Un sous-concept doit forcement partager les traits du concept père sinon il n'a pas sa place en tant que sous-concept.

« La différence avec le père » : Un sous-concept doit forcement avoir un ou plusieurs traits qui le différencient du concept père sinon le fait d'appartenir au sous-concept plutôt qu'au père n'apporte strictement aucune information et le sous-concept n'a dès lors pas de raison d'exister.

« La différence avec les frères » : Même principe que le point précédent. Si deux concepts frères sont identiques, les garder tous les deux n'apporte rien et ils doivent être fusionnés pour former un seul et même concept.

« La communauté avec les frères » : Des concepts frères partagent un même concept père et devraient donc forcement avoir des points communs hérités du père.

[6] bien que celui-ci soit déjà partiellement défini vu que l'attribut est attaché à une/des classe(s)
[7] A et B étant des classes de l'*ontologie*
[8] $A \setminus B$ étant inclus dans $A \cup B$, on a bien une facette plus restrictive que l'originale
[9] ou classe

Si les quatres principes ci-dessus ne sont pas respectés, nous sommes en droit de croire qu'il y a probablement un problème et une étude plus approfondie de la relation entre ces concepts devrait être faite.

2.3.3 Les différents langages

Quand on s'attarde un peu sur les *ontologies* sur le Web, il y a quelques grandes catégories de langages qui sont utilisés. Particulièrement : le *OWL* (en différentes versions).

OWL et ses différentes variantes

OWL est un langage d'*ontologie* porté par le *World Wide Web Consortium (W3C)* et se voulant être un standard dans le domaine. Il est basé sur *XML*. Il existe trois variantes de *OWL* :

OWL Description Logic (OWL-DL) La version « de base ». C'est la première version d'*OWL*. Elle se caractérise par plus de limitations que les langages plus anciens mais cela entraîne la particularité d'être décidable. Pour toute *ontologie* qui respecte les limites de *OWL-DL* on a la certitude qu'une requête formulée fournira toujours une réponse dans un temps fini.

OWL Full (OWL-F) Une version plus musclée que *OWL-DL*, elle lève les limitations et permet d'exprimer dans *OWL* ce qui n'est pas possible dans *OWL-DL* mais bien dans d'autres langages.

OWL Light (OWL-L) Au contraire de *OWL-F*, cette variante se veut plus légère et plus facile à utiliser que *OWL-DL*. Cependant il semblerait qu'elle soit en réalité tout aussi compliquée que *OWL-DL*.

Pour plus d'informations sur *OWL*, se reporter à [Horrocks, 2007, Horrocks *et al.*, 2003].

2.4 Moteur de recherche documentaire

2.4.1 Généralités

> « Un système de recherche d'information (RI) est un système qui permet de retrouver les documents pertinents à une requête d'utilisateur, à partir d'une base de documents volumineuse. »
>
> [Cao, 2004, p. 1]

Un moteur de recherche documentaire est un système qui permet à l'utilisateur d'exprimer une demande à laquelle le système répondra par une série de documents censés contenir les éléments de réponses qui pourront satisfaire l'utilisateur. Un moteur de recherche documentaire appartient donc à la catégorie des systèmes de *RI* tel que définis dans la citation ci-dessus. Deux concepts majeur ressortent. Il y a dans un premier temps le fait que l'utilisateur émet une demande, ce point sera traité à la section 2.4.2. Ensuite, le fait que le système va répondre en renvoyant un ensemble de documents qu'il

juge pertinent, ce qui sera traité à la section 2.4.3 et ce qui laisse sous-entendre que le système possède une collection de documents et un moyen de déterminer lesquels pourraient répondre à l'attente de l'utilisateur. Enfin, il y a un dernier concept qui n'est pas abordé dans la définition et sera traité à la section 2.4.4, il s'agit de la notion d'évaluation du système lui-même.

2.4.2 Notion de requête

On se basant sur [Cao, 2004], on peut définir une requête comme un besoin d'information de la part d'un utilisateur exprimé d'une manière telle que le système pourra calculer une réponse. Un exemple simple se trouve à la figure 2.11.

Dans [Cao, 2004], deux approches sont mises en avant en ce qui concerne la réalisation d'un système de *RI*.

Dans la première approche, l'approche qualifiée de « naïve », la requête se résume à une chaîne de caractères et le système va juste chercher après cette chaîne dans l'ensemble des documents qu'il connaît. Le principal avantage d'une telle méthode est la simplicité de la mise en place. Par contre, cela entraîne un temps de réponse important (surtout si la taille du corpus est importante) et que les requêtes n'ont pratiquement pas d'expressivités, on se contente de recherche la chaîne que représente la requête à l'identique.

La seconde approche consiste à utiliser un *index*. La requête peut dès lors être plus élaborée comme une expression booléenne entre différents termes que l'on veut présents ou absents. À côté de ce gain en expressivité, on gagne également en vitesse car parcourir l'*index* à la recherche des termes de la requête est bien plus rapide que de parcourir l'ensemble des documents un à un. Cependant, comme cela a déjà été abordé à la section 2.2.1, la création et la maintenance d'une telle structure a un coût en temps et en espace de stockage.

2.4.3 Notion de pertinence

En se basant toujours sur [Cao, 2004], on définit la notion de pertinence en disant qu'un document est pertinent si et seulement si l'utilisateur est en mesure de trouver l'information dont il a besoin dans ce document. Cette notion est considérée comme centrale dans un système de *RI*. Son but est de différencier les documents qui pourront aider l'utilisateur de ceux qui ne le pourront pas. Dès lors, on peut vite constater que la pertinence varie énormément d'un utilisateur à l'autre et même avec le même utilisateur, la pertinence peut varier. Il n'est donc pas possible de maîtriser parfaitement la pertinence

FIGURE 2.11 – Capture d'écran d'une requête dans un moteur de recherche documentaire

en toute généralité dans un moteur de recherche documentaire. Il faut donc se rabattre sur les points communs entre les différentes variations de la pertinence et essayer de voir comment identifier le plus possible le contexte et les attentes de l'utilisateur pour y ajuster la notion de pertinence au mieux.

En tenant compte du fait qu'on ne peut pas maîtriser la pertinence, on peut toutefois utiliser une approche basique qui est de considérer comme documents pertinents tous ceux qui possèdent les éléments de la requête[10]. Mais certains documents peuvent être plus pertinents que d'autres et on peut donc raffiner la notion de pertinence du système en calculant un indicateur pour chaque terme de la requête et pour chaque document qui tiendrait compte de la fréquence (normalisée) du terme dans le documents et de la généricité du terme. En effet, si un terme est présent dans 99% des documents, le fait qu'il soit présent dans un document de notre liste n'apporte pas beaucoup d'informations. Par contre, si un terme n'est présent que dans quelques rares documents, le fait de le trouver est très probablement bon signe. Par exemple, si on a un corpus de documents parlant de l'informatique, ajouter ou enlever les termes « ordinateur » ou « informatique » ne devrait pas avoir une grande influence sur les résultats renvoyés ou l'ordre de ceux-ci. Par contre, si on cherche un algorithme extrêmement précis qui a un nom qui ne fait référence à rien d'autre qu'à cet algorithme et qu'en plus cet algorithme n'est utilisé que dans un contexte extrêmement réduit, le fait de mettre le nom de cet algorithme ou non dans la requête va avoir une grande influence. Cette méthode de pondération utilise un indicateur qui s'appelle « *Term Frequence - Inverse Document Frequency (TF-IDF)* [11] ».

La figure 2.12 montre les documents les plus pertinents renvoyés par le moteur de recherche `Google` à la même requête que la figure 2.11.

2.4.4 Évaluation du système

Étant donné que le but d'un système de *RI* est de fournir à un utilisateur les documents pertinents par rapport à une demande qu'il aurait formulé, il semble important d'être en mesure d'évaluer le système pour voir s'il fournit des réponses de qualité ou non.

La méthode proposée par [Cao, 2004] consiste à calculer deux indicateurs — la précision et le rappel — sur la liste des documents retournée pour chaque requête extraite d'une liste représentative.

La précision est la proportion de documents pertinents (retournés) par rapport au nombre total de documents retournés. L'idée est de voir si le système ne renvoie pas trop de documents non pertinents. L'équation 2.5 montre la définition de la pertinence sous forme mathématique.

Le rappel, lui, est la proportion de documents pertinents retournés par le système par rapport au nombre total de documents pertinents se trouvant dans le corpus. L'idée est

[10] en considérant le cas simple où on ne spécifie pas qu'un terme doit être absent

[11] plus d'informations sur cet indicateur à l'annexe A qui se trouve page 87

de voir si le système n'oublie pas trop de documents pertinents. Il est également défini sous forme mathématique à l'équation 2.6.

Les deux indicateurs ne sont pas indépendants. Il est facile d'avoir un bon score dans l'un des deux indicateurs au détriment de l'autre : si on veut avoir une grande précision, on peut réduire énormément le nombre de documents retournés et alors ceux-ci auront plus de chance d'être pertinents mais cela fera baisser d'autant le rappel. À l'opposé, si on augmente le nombre de documents retournés, c'est le rappel qui va augmenter alors que la précision, elle, va diminuer. Il vaut donc mieux arriver à un compromis et avoir des valeurs moyennes dans les deux indicateurs plutôt que l'un très bon et l'autre très mauvais. D'après [Cao, 2004], un « bon » système tourne autour des 30% dans les deux indicateurs.

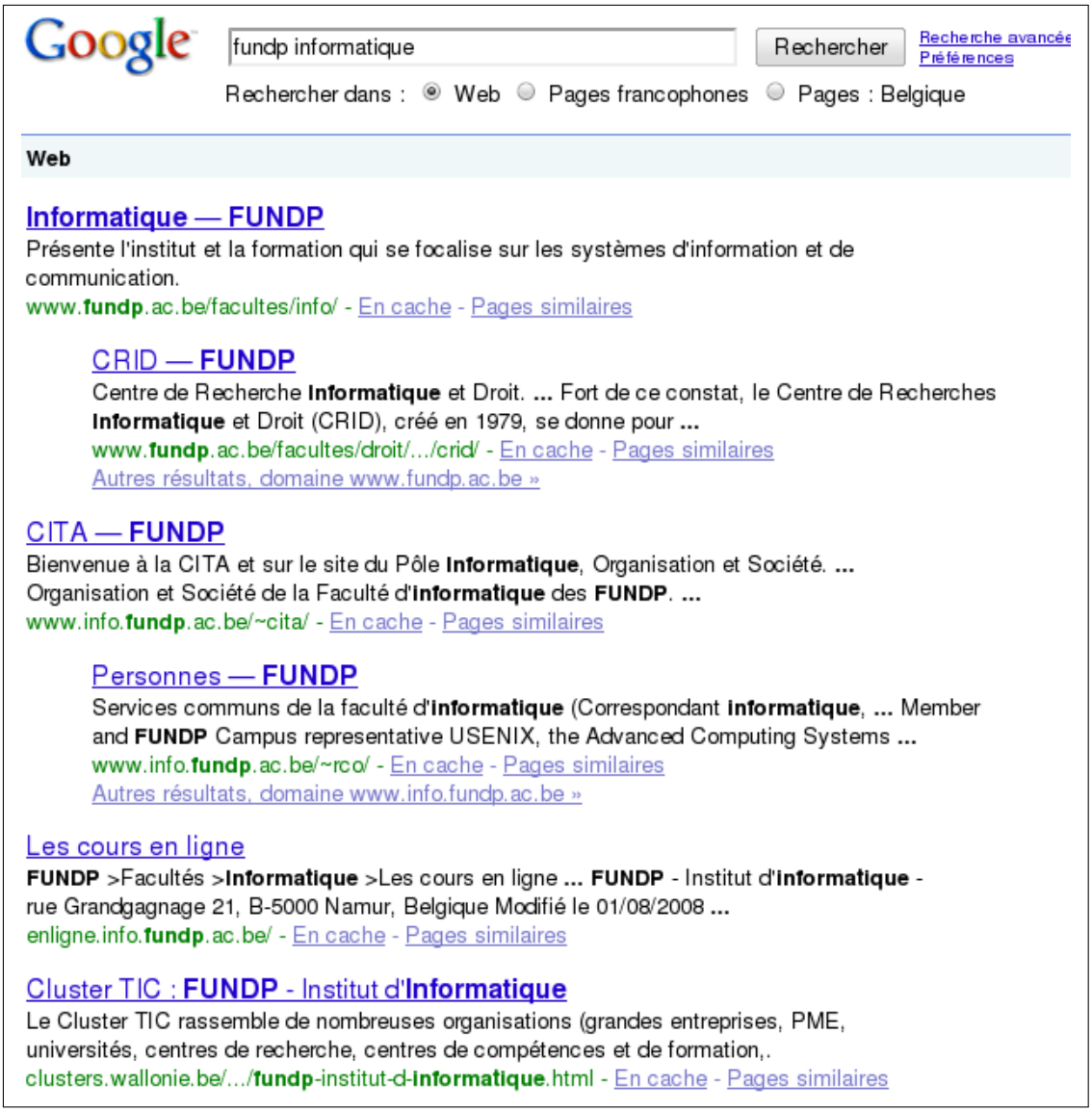

FIGURE 2.12 – Capture d'écran des résultats d'une requête dans un moteur de recherche documentaire

Pour évaluer le système, il faut donc d'abord choisir le corpus de documents sur lequel il va être évalué. Une fois cela fait, il faut préparer une série de requêtes. Et avoir des requêtes aussi variées que possible et essayant de couvrir la totalité du domaine du corpus. Après, il faut passer en revue tous le corpus et associés à chaque requête l'ensemble des documents pertinents (et donc par élimination les non pertinents aussi). Enfin, on lance chaque requête une par une et on note les documents qui sont retournés. Il ne reste qu'à calculer les deux indicateurs ci-dessous.

Comme on vient de le voire, le nombre de documents retournés a une influence directe sur les deux indicateurs. Donc, pour avoir une meilleure vue, on effectue la requête en limitant d'abord à un seul résultat[12], puis deux, puis trois et ainsi de suite jusqu'à avoir l'ensemble du corpus comme résultat. Pour chaque itération, on récupère une valeur pour la précision et pour le rappel qu'on peut alors reporter le tout sur un graphique qui après lissage sera similaire à celui de la figure 2.13 pour avoir une représentation de la qualité de notre système. Sur le graphique, on voit clairement que la diminution d'un des indicateurs se traduit par une augmentation de l'autre comme cela a été expliqué au paragraphe précédent.

$$D : L'ensemble\ des\ documents\ du\ corpus \tag{2.2}$$
$$P : \{d \in D | d\ est\ pertinent\} \tag{2.3}$$
$$R : \{d \in D | d\ est\ dans\ les\ résultats\ de\ la\ requête\} \tag{2.4}$$
$$Précision = \frac{P \cap R}{R} \tag{2.5}$$
$$Rappel = \frac{P \cap R}{P} \tag{2.6}$$

[12] qui sera dès lors le résultat le plus pertinent d'après le système

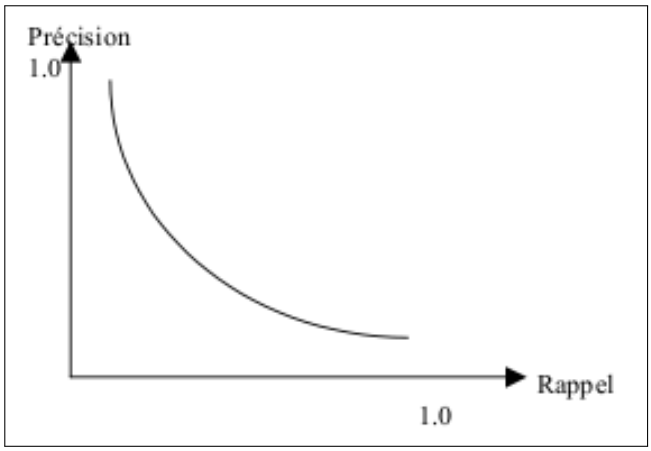

FIGURE 2.13 – Représentation graphique de l'évaluation d'un système de recherche d'information (extrait de [Cao, 2004])

Chapitre 3

Méthodologie

3.1 Introduction

Le but de ce chapitre est de définir les différentes méthodologies utilisées dans le reste de ce travail pour construire un moteur de recherche basé sur une *ontologie* et voir si ce système offre de meilleurs résultats qu'un moteur de recherche « classique ». Ce chapitre s'articule comme suit :

Construction de l'ontologie Le premier point abordé – à la section 3.2 – décrit la méthodologie qui sera suivie par la suite pour choisir le domaine de l'*ontologie*, faire le choix entre reprendre une *ontologie* existante ou en développer une et, si le choix est fait de construire une *ontologie*, quelle méthode sera suivie.

Construction du corpus Le deuxième point abordé – à la section 3.3 – décrit la méthodologie qui sera suivie par la suite pour constituer le corpus de documents. En commençant par définir le domaine et les restrictions sur celui-ci telle que la langue. Ensuite, on s'attaquera aux différents types d'emplacements où l'on ira chercher les documents et enfin à la façon de choisir les documents que l'on va prendre et ceux que l'on va laisser de côté.

Exploitation Le troisième point abordé – à la section 3.4 – met en avant la méthodologie qui sera suivie dans le développement de l'outil à savoir comment sera créé l'indexeur, comment sera ajoutée l'*ontologie* à celui-ci, le fait que l'*ontologie* sera une option et finalement comment sera créer le moteur de recherche exploitant ce qui aura été construit jusque là.

Évaluation Le quatrième et dernier point abordé dans ce chapitre – à la section 3.5 – mettra en avant les méthodes qui seront utilisées au chapitre 5 pour évaluer le système construit.

3.2 Méthodologie de construction de l'ontologie

Toute la méthodologie suivie pour le développement de l'*ontologie* repose sur [Noy *et al.*, 2005] tel qu'expliquée à la section 2.3.2 du chapitre précédent.

3.2.1 Choisir le domaine

La première chose à faire dans le développement d'une *ontologie* est bien évidement de définir le domaine qui sera couvert par celle-ci. Pour construire une *ontologie* sur un domaine spécifique, le mieux est d'avoir dans l'équipe des spécialistes de ce domaine, donc, en prenant le problème à l'envers, pour choisir le domaine, il semble bien de se demander les domaines pour lesquels on dispose de spécialistes. Pour aider à définir le domaine, on se reportera à **l'étape** 1[1].

Une fois le domaine choisi, il est important de définir les limites de l'*ontologie* et le niveau de détail. Pour ce faire, on se reportera de nouveau à **l'étape** 1[1] et principalement aux questions auxquelles on veut que l'*ontologie* soit capable d'apporter des réponses.

3.2.2 Chercher une ontologie existante

Pour la suite, on passe logiquement à **l'étape** 2[1] qui consiste à chercher dans les *ontologie*s existantes s'il y en a une qui correspond à ce que l'on cherche à construire. L'article [Noy *et al.*, 2005] propose une série d'*Uniform Resource Locator (URL)* ce qui semble donc être une bonne piste pour commencer à chercher.

3.2.3 Construire sa propre ontologie

S'il n'existe pas d'*ontologie* ou si celle-ci ne correspond pas complètement à ce dont on a besoin, il faudra aménager l'*ontologie* ou en créer une depuis zéro. Pour ce faire, on va se baser sur **l'étape** 3[1] et faire une liste la plus complète possible de tous les termes du domaine ciblé par notre *ontologie*. Après, sur base de **l'étape** 4[1], on va créer la *taxonomie* des concepts en attachant une attention toute particulière à la remarque de [Bachimont, 2000] pour essayer de trouver le bon moment pour différencier un concept en plusieurs sous-concepts.

On va ensuite pouvoir enrichir les concepts avec des attributs et des relations en suivant **l'étape** 5[1] et caractériser le tout avec des facettes comme expliqué à **l'étape** 6[1]. Lors de ces deux étapes, on fera en sorte de ne pas mettre en place d'élément d'une complexité telle qu'il ferait basculer l'*ontologie* dans le non-décidable. Autrement dit, il faut que l'*ontologie* soit exprimable en restant dans les limites imposées par *OWL-DL* et qu'on ne soit donc pas obligé de tenir compte de l'éventualité qu'une requête effectuée sur l'*ontologie* boucle à l'infini.

Pour l'utilisation de l'*ontologie* par la suite, on a besoin d'un attribut présent dans chaque instance et qui sera donc placé dans le concept racine de l'*ontologie*. Cet attribut comportera la liste des termes que l'on peut trouver dans un document et qui fait référence à l'instance en question.

[1]Les différentes étapes sont celles extraites de la méthodologie de [Noy *et al.*, 2005]. Elles sont expliquées à la section 2.3.2 (page 31, chapitre 2)

Une fois la structure de l'*ontologie* terminée, on peut passer à **l'étape** 7[1] et créer des instances de l'*ontologie* sur les sujets pour lesquels on veut que l'*ontologie* puisse trouver des documents. On apportera une attention toute particulière à renseigner tous les termes pour l'attribut dont on a parlé au paragraphe précédent vu que c'est ce qui sera utilisé par le moteur de recherche documentaire pour trouver les documents pertinents.

3.3 Méthodologie de construction du corpus

3.3.1 Choisir et restreindre le domaine

La première chose à faire quand on parle de la construction du corpus c'est de se mettre d'accord sur le(s) domaine(s) des documents qui composeront ce corpus. Le choix du domaine ici sera de reprendre le même domaine que celui de l'*ontologie*.

Une fois le problème du domaine réglé, le suivant sur la liste est le problème de la langue. Comme expliqué dans la partie analyse des documents de la section 2.2.2. Pour le présent travail, on va se concentrer sur les documents en français et donc essayer d'éviter tout document rédigé dans une langue autre dans le corpus.

3.3.2 Emplacement de la collecte

Comme nous avons vu à la partie collecte des documents de la section 2.2.2, il est possible de composer son corpus en collectant des documents depuis des sources très variées. Dans le cadre de l'outil, on ira principalement sur `internet` en commençant par exploiter le moteur de recherche de `Yahoo` qui met à disposition une *Application programming interface (API)* basique d'interrogation à l'utilisation gratuite.

3.3.3 Critères de sélection des documents

Dans tous les documents que nous allons collecter, il y aura probablement des documents qui ne nous conviennent pas. Dans un premier temps, il y a tous ceux qui sont dans un format qui n'est pas supporté par l'outil. Ensuite, même si le format est connu, il y a quand même un taux d'erreur. Donc dans les faits, on va déjà éliminer tous les documents que l'outil n'est pas capable de gérer.

Ensuite, dans les documents que l'outil est effectivement capable d'analyser, il faut éliminer tous les documents qui ne sont pas rédigés en français conformément à ce qui a été dit à la section 3.3.1. Pour éliminer ces documents, on ne peut pas se contenter d'un filtre comme expliqué à la section 2.2.2 vu qu'ici on doit regarder le contenu pour déterminer la langue du document. Il faudra donc laisser l'outil extraire le contenu du document, appliquer une méthode pour déterminer la langue du document et n'analyser que les documents correspondant au français.

3.4 Méthodologie d'exploitation du corpus

L'exploitation du corpus repose sur l'utilisation d'outils. La méthodologie qui sera suivie pour le développement des outils est de commencer par construire un système d'*indexation* – section 3.4.1 – puis d'y ajouter l'*ontologie* construite précédemment – section 3.4.2 – avec la possibilité de couper le lien et de finir par la création d'un moteur de recherche – section 3.4.3 – permettant d'exploiter l'*index* et l'*ontologie*.

3.4.1 Création d'un index

La brique de base c'est la construction de l'*index* et, plus globalement, l'ensemble du processus d'*indexation* tel qu'expliqué à la section 2.2.2.

Le premier niveau c'est la collecte des documents. On a vu à la section 3.3.2 que la source principale qui allait être utilisée est `Internet` et particulièrement l'*API* du moteur de recherche `Yahoo`. Il faut donc au minimum deux collecteurs pour gérer ces deux cas.

Le deuxième niveau comporte les filtres. On n'a pas relevé de critère simple pouvant être utilisé pour filtrer les documents, il n'y a donc à priori pas besoin de s'attarder sur ce niveau dans l'implémentation.

Le troisième niveau est la détection du format du document. Pour ce faire, la méthode la plus simple est de regarder l'extension du fichier et de comparer avec une liste interne.

Quatrième niveau, les *parsers*. Étant donné que ce n'est pas l'objectif premier de ce travail, l'utilisation de librairies existantes sera préconisée pour cette partie. Concernant les types de format à gérer, vu que l'on va collecter les documents depuis internet, il semble qu'un *parser* de page `Web` est inévitable. De même, on prévoira un *parser PDF* vu l'utilisation répandue de ce format.

Niveau cinq, une fois le contenu extrait, on peut passer à l'analyse en commençant par la détection de la langue. Vu le choix fait à la section 3.3.3 de ne traiter que les documents de langue française, si la détection de langue retourne autre chose, on considère juste le document comme erroné et on arrête le traitement là.

Enfin, le dernier niveau consiste à écrire dans un *index* les informations relatives aux documents. On utilisera pour ce faire la librairie *Lucene*.

Si on reprend le schéma de la figure 2.8 et qu'on l'adapte avec ce qui vient d'être dit, on obtient la figure 3.1 .

3.4.2 Couplage de l'index avec l'ontologie

Comment mettre en place le couplage ?

La méthode la plus simple pour coupler l'*index* avec l'*ontologie* est de regarder pour chaque terme que l'on va écrire dans l'*index* les concepts de l'*ontologie* auquel ce terme est associé.

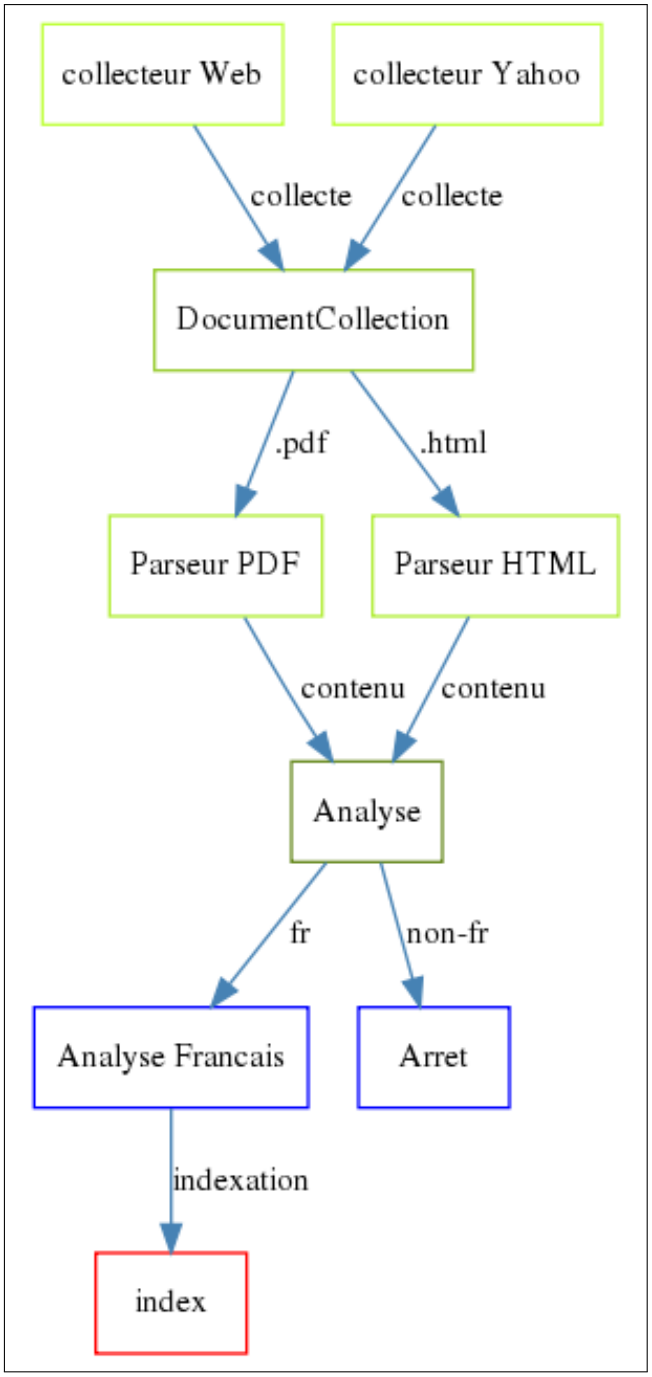

FIGURE 3.1 – Illustration du processus d'indexation tel qu'il devra être

Possibilité de comparaison

Pour pouvoir estimer les performances de notre système il faut pouvoir le tester. Pour ce faire, on s'assurera de pouvoir faire des recherches avec ou sans l'*ontologie* ce qui nous permettra de faire des comparaisons des deux systèmes. La méthode la plus simple consiste à générer l'*index* une fois avec les informations extraites de l'*ontologie* et une deuxième fois sans mais cela prendrait beaucoup de temps et d'espace. La méthode qu'on mettra en application consiste plutôt à bien identifier les éléments en provenance de l'*ontologie* d'une part et les autres d'autre part et de s'assurer qu'elles restent bien séparées.

3.4.3 Création du moteur de recherche documentaire

Une fois l'*index* et l'*ontologie* créés, il reste encore à mettre au point un outil permettant de les exploiter. Il faut donc un système capable de lire et exploiter un *index* au format *Lucene* de même qu'une *ontologie* au format *OWL-DL*. De même, pour pouvoir évaluer le système d'*indexation*, on a décidé de bien séparer les informations propres à l'*ontologie* du reste et on veillera donc dans le moteur de recherche documentaire à pouvoir restreindre l'utilisation de l'*index* aux informations classiques.

3.5 Méthodologie d'évaluation du système

La méthode d'évaluation qui sera utilisée pour notre système est celle décrite à la section 2.4.4 . À savoir qu'on va préparer une série d'au moins dix requêtes portant sur le domaine du corpus que l'on aura constitué préalablement. À partie de là, on pourra évaluer notre système par rapport à lui-même en regardant la différence de résultats avant ou sans l'*ontologie*, comme expliqué à la section 3.5.1. Et dans un second temps, on pourra évaluer notre système par rapport à un autre moteur de recherche plus « classique », ce qui sera expliqué à la section 3.5.2.

Une petite différence est à noter sur la méthodologie d'évaluation qui sera suivie. Calculer le rappel nécessite de posséder une liste des documents pertinents pour chaque requête réalisée par une autre méthode. Nous ne disposons pas d'une telle liste et la construire demanderait trop de temps. Au lieu de calculer le rappel sur l'ensemble du corpus, on se contentera donc de le calculer sur l'ensemble des documents retournés par les différentes exécutions des différents moteurs de recherches documentaires.

3.5.1 Évaluation interne

La première forme d'évaluation consiste à faire des tests avec l'*ontologie* puis de refaire les mêmes tests sans l'*ontologie* et de comparer les résultats.

On soumettra chaque requête au moteur de recherche en spécifiant qu'on désire utiliser l'ontologie et qu'on ne veut que le résultat le plus pertinents. On fera la même chose sans utiliser l'ontologie. On calcule ensuite la précision et le rappel.

On recommence l'opération en demandant les deux résultats les plus pertinents, puis les trois et ainsi de suite jusqu'à demander autant de documents que le corpus en contient.

On pourra alors tracer les graphiques de précision/rappel pour les deux versions du système et tenter de déterminer ce qu'apporte l'*ontologie*.

3.5.2 Évaluation externe

La deuxième forme d'évaluation consiste, elle, à comparer les résultats de notre système avec ceux d'un moteur de recherche existant pour voir comment on se situe par rapport à lui.

Lors de la première forme d'évaluation, on aura executer un certain nombre de requête sur notre système avec l'*ontologie* activée, il ne nous reste donc ici qu'à soumettre les mêmes requêtes sur un autre moteur de recherche et à comparer les résultats.

Chapitre 4

Les outils

4.1 Introduction

La partie développement de ce travail se compose de plusieurs outils prévus pour interagir entre eux et construits de manière incrémentale. Le présent chapitre a pour but de décrire chaque « niveau » de cette hiérarchie d'outils et les liens qui existent entre chaque niveau :

Indexeur La brique de base est l'indexeur de document qui se trouve donc tout au pied de la hiérarchie. L'indexeur reprend les différentes phases identifiées pour un processus d'*indexation* et explique ce qui a été fait pour chaque phase. Cette partie sera traitée à la section 4.2.

Ontologie La deuxième partie du développement est la conception d'une *ontologie* pour un domaine spécifique et d'une manière telle que l'on puisse en tirer profit dans un moteur de recherche documentaire sur ce même domaine. Cette partie est traitée à la section 4.3.

Couplage Indexeur et ontologie Une fois l'indexeur et l'*ontologie* terminés, il faut les faire travailler ensemble. La section 4.4 montre comment cela a été réalisé dans le cadre de ce travail.

Moteur de recherche La dernière brique de ce travail est le moteur de recherche qui a pour but d'utiliser ce qui a été fait précédemment pour qu'un utilisateur puisse trouver l'information qu'il désire. La plus grosse partie de cet outil réside dans les interfaces graphiques et dans la façon dont il utilise ce qui a été fait précédemment. Tout cela sera traité à la section 4.5.

4.2 Indexeur de documents

Comme dit dans l'introduction de ce chapitre, l'outil de base, celui sur lequel repose tout le reste du travail est l'indexeur de documents. C'est sur lui que l'on va greffer les nouvelles fonctionnalités et c'est le résultat généré par cet outil qui sera utilisé comme ressource dans le moteur de recherche.

L'indexeur a été développé en *Java*. Il se compose de tous les processus décrits à la section 2.2.2 traitant de l'*indexation* de document de type texte. On va donc reprendre chacun de ces processus un par un et expliquer ce qui y correspond dans l'outil.

4.2.1 Processus de collecte

Le fonctionnement de base des collecteurs dans notre outil repose dans un premier temps sur la spécification de deux points importants :

1. Où aller chercher les documents ?

2. Comment décider quand s'arrêter ?

Après, le travail du collecteur est d'appeler le constructeur de l'objet représentant un document et de l'ajouter dans une liste des documents à traiter. Cette étape supporte déjà un premier tri, tous les candidats documents récupérés par un collecteur pour lesquels on est incapable d'extraire des informations de base comme l'extension ou n'étant pas accessibles du tout[1] par l'outil sont éliminés ici.

L'image B.1 est une capture d'écran du processus de collecte de l'indexeur. On peut voir sur cette image trois graphiques. Le premier, en haut à gauche, représente la répartition des types de documents en pourcentage du nombre total de documents. Le deuxième graphique, en haut à droite, montre le premier tri des documents, il va montrer le pourcentage de documents collectés qui seront refusés à ce niveau et ne seront donc pas traités. Enfin, le troisième et dernier graphique, celui du bas, montre pour chaque collecteur actif le nombre de documents collectés. Ces trois graphiques sont mis à jour en temps réel pendant l'exécution de cette phase par l'indexeur.

Les collecteurs sont des éléments indispensable au bon fonctionnement de l'indexeur. Cependant, ils ne jouent pas un rôle central dans l'objectif de ce travail dans le sens où les collecteurs n'ont pas le moindre lien avec l'*ontologie*. Pour cette raison, la création de ces collecteurs n'a pas été réalisée par moi mais bien par Fabrice `Estiévenart`.

FileCollector

Un premier collecteur de documents est le *FileCollector*. Son but est de parcourir un disque dur de la machine sur laquelle est lancée l'outil. Il prend en compte deux paramètres.

Pour le premier paramètre, il s'agit de la cible du collecteur. On peut spécifier directement un fichier ou alors un répertoire. Si le collecteur a pour cible l'adresse d'un répertoire, il va alors considérer l'ensemble des fichiers et répertoires inclus dans celui-ci.

[1] Un exemple simple de problème d'accessibilité est d'obtenir l'adresse d'un fichier sur une page `Web` d'un intranet mais ne pas avoir la permission d'accéder au fichier en lui-même.

TABLE 4.1 – La liste des collecteurs présents dans l'outil

Nom	Description	Critères
`FileCollector`	Recherche dans un répertoire locale	Choix de parcourir ou non les sous-répertoires.
`YahooCollector`	Utilise l'*API* de `Yahoo` pour interroger son moteur de recherche	Utilise un paramètre pour spécifier le nombre maximum de résultats à renvoyer
`Web-Crawler`	Recherche sur le `Web`	On fournit le point de départ, le nombre de lien consécutifs que le collecteur peut suivre et s'il doit rester ou non dans le domaine d'origine.
`RSSCollector`	Recherche dans les pages cibles d'un flux *RSS*	Utilise un paramètre pour spécifier le nombre maximum de résultats à renvoyer.

Le second paramètre du collecteur est un booléen qui donne ou non le droit au collecteur d'agir de manière récursive. Cela n'a de sens que si on donne un répertoire comme cible. Dans ce cas, le collecteur récupère le contenu du répertoire y compris d'autres répertoires s'il y en a. Si le booléen autorise la récursivité, le collecteur va alors recommencer son travail à l'intérieur de ses sous-répertoires. Si, par contre, le booléen interdit la récursivité, il se contentera des fichiers du répertoire cible et ne regardera pas dans les sous-répertoires éventuels.

En ce qui concerne le problème d'arrêt, les disques durs possèdent un nombre limité de fichiers, le collecteur finira donc pas s'arrêter même si la récursivité est activée.

YahooCollector

Le deuxième collecteur de documents est le *YahooCollector*. Ce collecteur utilise l'*API* du moteur de recherche `Yahoo`. On soumet une requête et on reçoit en retour une liste d'*URL*. On utilise deux paramètres de l'*API* en question.

Le premier paramètre est la requête que l'on veut soumettre au moteur de recherche. On formule la requête comme on le ferait en utilisant directement le moteur de recherche lui-même.

Le second paramètre de l'*API* utilisé est le nombre maximum de résultats souhaités. Ce paramètre est donc très utile pour garantir que le collecteur ne renverra qu'un nombre fini de documents et, pour peu que le moteur de recherche de `Yahoo` ne boucle pas infiniment sur la requête, notre collecteur finira par s'arrêter également.

Web-Crawler

Le troisième collecteur de documents de l'outil est le *Web-Clawler*. Comme son nom l'indique, il s'agit d'un *crawler* qui parcourt le Web. Il prend trois arguments.

Le premier argument est le point de départ du collecteur. On lui donne une *URL* qu'on décompose en deux. D'un côté, il y a l'adresse du domaine où se trouve le point de départ que l'on souhaite et de l'autre le chemin relatif dans ce domaine pour trouver ce point de départ. Si on recombine les deux parties, on retrouve donc bien l'*URL* complète du point de départ souhaité. Le choix de décomposer l'*URL* ainsi a été fait pour simplifier les tests relatifs à l'argument suivant.

Le deuxième argument est un booléen qui définit si le collecteur est autorisé à suivre des liens qui le feraient sortir du domaine de départ ou s'il doit absolument rester dans celui-ci. Pour illustrer cet argument, prenons une page du site Web des *Facultés Universitaires Notre-Dame de la Paix (FUNDP)*. Sur celle-ci, supposons qu'il y a trois liens hypertexts. Les deux premiers pointent vers deux autres pages du site des *FUNDP* et le troisième pointe, lui, vers le site de la ville de *Namur*. Si le booléen autorise à quitter le domaine, le *crawler* va s'exécuter récursivement sur les trois liens. Par contre, si le booléen oblige le *crawler* à rester dans le domaine, il ne s'exécutera récursivement que sur les deux premiers liens et ignorera le troisième.

Enfin, le troisième et dernier argument spécifie la profondeur maximale de récursivité. Pour revenir à l'exemple précédent, si cet argument est à zéro quand on arrive, il va détecter les trois liens sur la page mais n'en suivra aucun. S'il est supérieur à zéro, il suivra le lien (en respectant ou non le domaine suivant la valeur de l'argument deux) et s'appellera récursivement en décrémentant la valeur de l'argument trois d'une unité. S'il est toujours supérieur à zéro, il s'appellera récursivement sur les liens éventuels de cette page jusqu'à ce qu'il finisse par avoir une valeur nulle qui arrêtera le processus.

Étant donné que le nombre d'appels récursifs est limité, on n'est pas obligé de se préoccuper de la présence éventuelle de cycle dans les pages[2]. Cependant, le traitement d'une page au niveau du collecteur n'étant pas forcement sans coût, un mécanisme de cache a été mis en place qui vérifie d'abord qu'une page n'a pas déjà été traitée. De plus, avoir plusieurs fois la même page dans le collecteur donnerait deux documents identiques qui seraient traités tous les deux et écrits tous les deux dans l'*index* ce qui d'un premier côté ferait perdre du temps et des ressources et, dans un second temps, pourrait entraîner un biais dans l'utilisation de l'*index* par la suite. Autant donc s'assurer dès le début de ne pas créer de doublons inutiles et donc de s'assurer que la page qu'on s'apprête à examiner n'a pas déjà été visitée plus tôt.

RSSCollector

Le quatrième et dernier collecteur est le *RSSCollector*. Son but est de consulter un flux *RSS*. Il prend en compte deux arguments.

[2]Une série de lien permettant de partir d'une page et d'y revenir par la suite

TABLE 4.2 – La liste des filtres implémentés dans l'outil

Nom	Description
`SizeFilter`	Ce filtre prend en argument un nombre de `bytes` et rejette tous les documents ayant une taille supérieur à cet argument
`SimpleUrlFilter`	Ce filtre prend en argument une chaîne de caractères et un `booléen` et accepte ou rejette en fonction du `booléen` tous les documents qui contiennent la chaîne de caractères passée en argument.
`RegExpFilter`	Ce filtre fonctionne comme le précédent mais en se basant sur une expression régulière

Le premier argument du collecteur est l'adresse du flux *RSS* qu'il doit consulter. À partir de là, en considérant que le flux est bien construit, chaque élément qui le compose possède un lien vers une page `Web`. Le collecteur va dès lors récupérer ces pages et les renvoyer comme résultat.

Le second argument est une limite de résultat. C'est à dire que le collecteur n'évaluera pas forcement tous les éléments du flux. S'il le nombre d'éléments est supérieur à la valeur de l'argument il se contentera d'évaluer les éléments dans l'ordre jusqu'à atteindre cette valeur puis il s'arrêtera et renverra l'ensemble des pages qu'il aura collecté jusque là. Cet argument étant obligatoire, le collecteur finira toujours par atteindre cette valeur ou s'arrêtera même avant s'il n'y a pas assez d'éléments dans le flux *RSS*. Dans un cas comme dans l'autre, le collecteur s'arrête.

4.2.2 Processus de filtrage

Comme expliqué à la section 3.4.1 du chapitre sur la méthodologie, dans notre cas particulier, on a pas détecté la nécessité de mettre en place des filtres. Cependant certains filtres basiques ont quand même été développé.

Le fonctionnement des filtres est inspiré d'un mécanisme de filtrage de la librairie *Lucene* qui permet la mise en place de filtre sur le flux utilisé pour indexer le contenu des documents et éliminer de celui-ci les termes[3] qui ne répondraient pas à certains critères.

Le filtre *SimpleUrlFilter* a été réalisé par Fabrice `Estiévenart`. Les deux autres filtres ont été réalisé par mes soins.

SizeFilter

Le premier filtre présent dans l'outil est le *SizeFilter*. Son but est d'éliminer tous les documents qui dépassent un certain poids. Il prend donc un nombre représentant la taille

[3]En réalité, il s'agit de « token » qui consistent en une série d'information dont la première est le terme, suivi de deux entiers indiquant sa position dans le texte

limite comme argument. Après, le filtre demande la taille du fichier et compare avec la valeur de son argument.

SimpleUrlFilter

Le deuxième filtre est le *SimpleUrlFilter*. Son but est de regarder après une certaine chaîne de caractères dans l'*URL* des documents. Ce filtre prend deux arguments.

Le premier argument est la chaîne de caractères. Le filtre va ensuite regarder dans l'*URL* de chaque document si cette chaîne est présente ou non. Le choix d'éliminer ou non le document ne peut pas être fait avec juste cet argument.

Le second argument de ce filtre est un booléen qui décide si l'on recherche les documents qui possèdent la chaîne du premier argument ou si au contraire, ce que l'on recherche ce sont les documents qui ne possèdent pas cette chaîne.

RegExpFilter

Le dernier filtre qui a été développé dans l'outil est le *RegExpFilter*, qui, comme son nom l'indique est basé sur l'utilisation d'une *RegExp*. Il fonctionne exactement comme le filtre précédent excepté le fait que l'on ne recherche plus une chaîne de caractère précise mais toute chaîne respectant un certain *pattern* représenté par la *RegExp* passée en argument.

Comme pour le filtre précédent, on peut demander à garder les documents qui possèdent le *pattern* ou au contraire les exclure et ne garder que les documents qui ne possède pas ce *pattern*.

4.2.3 Processus de parsing

Les différents *parsers* développés dans le cadre de l'outil reposent sur des librairies ou des *API* existantes[4]. Chaque *parser* est nommé en fonction du format qu'il permet d'exploiter et de la librairie/*API* sur laquelle il repose. Mon travail dans le cadre du processus de *parsing* a été d'exploiter les librairies existantes pour extraire le contenu des documents et pouvoir traiter ce contenu de manière uniforme par la suite.

L'image B.2 est une capture d'écran du processus de *parsing* de l'indexeur. On peut y voir trois graphiques. Le premier, en haut à gauche, représente la répartition des documents entre les différents *parsers*. Le deuxième, en haut à droite, montre la répartition des documents en fonction de leur statut (en attente, traité, pas traité pour cause d'erreur, ...). Enfin, le troisième et dernier graphique, en bas, représente le nombre de documents traités en fonction du temps que cela a pris et ce par *parser*.

[4]Toutes les librairies et *API* utilisées sont soit intégrées directement dans la distribution de *Java* 1.6 soit distribuées sous licence *libre*

PDFBoxParserPDF

Le premier *parser* présent dans l'outil utilise la version 0.7.3 de la librairie PDFBox. Comme son nom l'indique, ce *parser* permet d'extraire le contenu des documents au format *PDF*. Si les méta-données comme le titre, le sujet, l'auteur sont présents[5], la librairie les détecteras et se chargera de les extraire.

SaxParserXML

Ce deuxième *parser* a été mis en place pour gérer les documents au format *XML*. Il utilise *Simple API for XML (SAX)* qui est présente directement dans la distribution *Java*. Il permet donc de récupérer le contenu des fichiers *XML* mais il peut également être utilisé pour traiter des documents au format *HyperText Markup Language (HTML)* même si cela n'est pas conseillé. En effet, les pages *HTML* comportent souvent des fautes[6] qui produiraient une erreur avec ce *parser*. Et en plus, ce *parser* ne cherche pas après des méta-données. Pour ces deux raisons, il vaut mieux utilisé le *NekoParserHTML* pour les documents au format *HTML*.

POIParserWord et POIParserPPT

Les différents formats utilisé par la suite *MS* Office peuvent être consulté avec la librairie *Poor Obfuscation Implementation (POI)* Apache développée par la Fondation Apache. La version utilisée dans l'outil est la 3.5.1 beta 1. Cette dernière ne supporte pas la version 2007 et supérieur des formats de la suite bureautique de *MS* mais il était annoncé que ces formats seraient pris en charge prochainement. Pour plus d'information, voir [Oliver *et al.*, 2008].

Dans le cadre de l'outil, on a mis en place deux *parsers*. Un pour les documents textuel de *Word* et un autre pour les documents de présentation de *Powerpoint*. Les documents réalisés à l'aide du tableur *Excel* ont été laissés de côté. Ces deux *parsers* ne récupèrent pas les méta-données éventuellement présentes dans les documents.

ParserTXT

Ce *parser* n'en est pas vraiment un. Il est employé sur les documents qui ne sont pas du tout structuré aussi appelé texte brut. Il n'effectue aucun traitement sur le contenu du fichier et le retourne tel quel.

ParserRTF

Le *ParserRFT* est un *parser* qui utilise l'*API* fournie directement dans la distribution de base de *Java* pour extraire le contenu des documents au format *Rich Text Format (RTF)*. Il n'y a pas de méta-données récupérables dans ce format.

[5] Ces données sont renseignées dans le présent document sous sa forme électronique. Elles peuvent être consultées en affichant les propriétés du document dans le visionneur

[6] Une faute courante dans un fichier *HTML* est d'oublier une balise fermante ce qui dans le cas présent fausserait complètement le résultat du *parser*

TABLE 4.3 – La liste des parsers disponibles dans l'outil

Nom	Format(s) supporté(s)	Librairie
PDFBoxParserPDF	*PDF*	la librairie externe PDFBox en version 0.7.3
SaxParserXML	*XML* et *HTML*	la librairie *SAX* de *Java*
POIParserWord	*Word*	la librairie *POI* Apache en version 3.5.1 beta 1
POIParserPPT	*Powerpoint*	
ParserTXT	texte brut	
ParserRFT	*RTF*	librairie interne à *Java*
NekoParserHTML	toute la famille des formats *HTML*	La librairie externe NekoHTML en version 1.9.9
JOpenDocumentParserODT	*ODT*	JOpenDocument en version 1.1

NekoParserHTML

Le *parser* suivant repose sur la librairie NekoHTML en version 1.9.9 qui a pour but d'analyser les documents *HTML* et d'en extraire le contenu. Il est plus robuste que le *SaxParserXML* car il intègre une tolérance aux erreurs de syntaxe dans les documents qu'il analyse. Cette librairie permet de gérer jusqu'à la version 4 de *HTML*. Le *parser* va chercher dans le document le titre de celui-ci via les balises « `<title>` » et « `</title>` ».

JOpenDocumentParserODT

Le dernier *parser* utilise une librairie relativement récente pour extraire le contenu des documents de la suite *Libre* OpenOffice.Org. Il s'agit de la version 1.1 de la librairie JOpenDocument. La librairie ne permet pas encore de récupérer les méta-données éventuellement présentes dans les documents. Plus d'information sur [jOp, 2008].

La librairie permet de lire et écrire les documents textes et les documents du tableur mais seul les documents au format *OpenDocument Text (ODT)* sont gère par l'outil par un *parser*.

4.2.4 Processus d'analyse

Une fois le *parsing* terminé, on a accès aux contenus des documents de manière exploitable. Il est alors temps d'examiner le contenu de chaque document pour voir lesquels sont en français et ne garder que ceux là pour la suite. L'algorithme utilisé calcul une probabilité pour chaque langage qu'il connaît sur base du nombre de *stopwords* de la langue en question présent dans le document sur le nombre total de termes.

L'algorithme utilisé a été rédigé par Fabrice Estiévenart.

Une fois les documents passés en revue pour ne garder que ceux identifié comme étant des documents en français, on peut faire l'analyse à proprement parlé. Si on se concentre sur l'indexeur classique, cette analyse se résume à éliminer tous les *stopwords* qui ne

nous intéressent plus une fois la langue détectée. Cela se fait très facilement en utilisant la librairie *Lucene*. La librairie utilise un buffer pour temporiser l'écriture dans l'*index* et fournit un mécanisme pour rajouter des filtres à ce buffer pour en nettoyer le contenu et fourni d'ailleurs une série de filtres prédéfinis. La librairie fournit en particulier un filtre pour les *stopwords* de la langue française.

4.2.5 Processus d'écriture

Toute la partie qui correspond au processus d'écriture de l'*index* repose entièrement sur la librairie *Lucene*. Il ne reste qu'à remplir un objet compréhensible par la librairie avec toutes les informations sur les documents que l'on souhaite voir présente dans l'*index*. La structuration des données se fait sous forme de champs. Dans l'outil, on a des champs pour le titre, l'auteur, l'*URL*, la date de dernière modification et le contenu. Pour chacun de ses champs, il nous reste à préciser à la librairie la façon dont on veut qu'ils soit stockés. Pour le contenu, on veut clairement qu'il soit présent sous la forme d'un *index*, pour les autres champs, on a plutôt besoin de la valeur complète.

Quand on a fini de spécifier ce que l'on voulait à la librairie, il ne reste plus qu'à lui dire d'écrire le contenu du buffer dans l'emplacement qu'on aura spécifié préalablement pour l'*index*.

L'image B.3 est une capture d'écran du processus d'écriture de l'*index* dans l'indexeur. L'écriture de *index* par la librairie *Lucene* se résume vraiment à la toute dernière étape et se fait dans un temps inférieur à la milliseconde pratiquement dans tous les cas ce qui rend ce graphique bien moins intéressant que ceux des phases précédentes.

4.3 Ontologie

Une fois l'indexeur « classique » terminé, on peut alors aborder l'*ontologie*. Le but de celle-ci est de venir s'intégrer dans l'indexeur déjà réalisé et dans le moteur de recherche documentaire à venir.

4.3.1 Association de termes à des concepts

La première chose à prendre en compte dans le développement de l'*ontologie* est le but poursuivi dans ce travail. Il a été décidé au chapitre 3 que l'on utiliserait l'*ontologie* en cherchant les concepts de celle-ci associés à un terme particulier.

Il semble donc que la première chose à mettre dans notre *ontologie* est une structure pouvant supporter cette association de termes à des concepts.

On va donc créer un attribut – appelé *Data Property* dans *OWL* – qui aura pour domaine « `Thing` » et qui aura comme valeur une chaîne de caractère. « `Thing` » étant la racine[7] de la hiérarchie de concepts de l'*ontologie*, tous les autres concepts définis hériterons de cet attribut.

[7] On peut comparer « `Thing` » de *OWL* avec la classe « `Object` » de *Java*.

En ce qui concerne les cardinalités, l'attribut pourra prendre un nombre quelconque éventuellement nul de valeur. Un concept sans termes associés ne serait pas d'une grande utilité pour l'utilisation que l'on souhaite faire de l'*ontologie* mais par un soucis de généralité et réutilisabilité, il vaut mieux ne pas forcer la présence de valeur pour cet attribut.

On obtient donc le *Data Property* « terme » tel que représenté à la figure 4.1 qui se trouve à la page 56.

4.3.2 Choisir le domaine

Comme évoqué à la section 3.2.1, quand on veut se lancer dans la création d'une *ontologie*, il faut commencer par bien spécifier le domaine. Le but de ce travail étant de montrer qu'on peut améliorer un moteur de recherche documentaire dans un domaine particulier, le choix de ce domaine pour l'étude de cas n'a pas d'influence sur les résultats. Le choix du domaine est donc principalement dicté par mon expertise personnelle, les documents disponibles et l'utilisation possible du résultat.

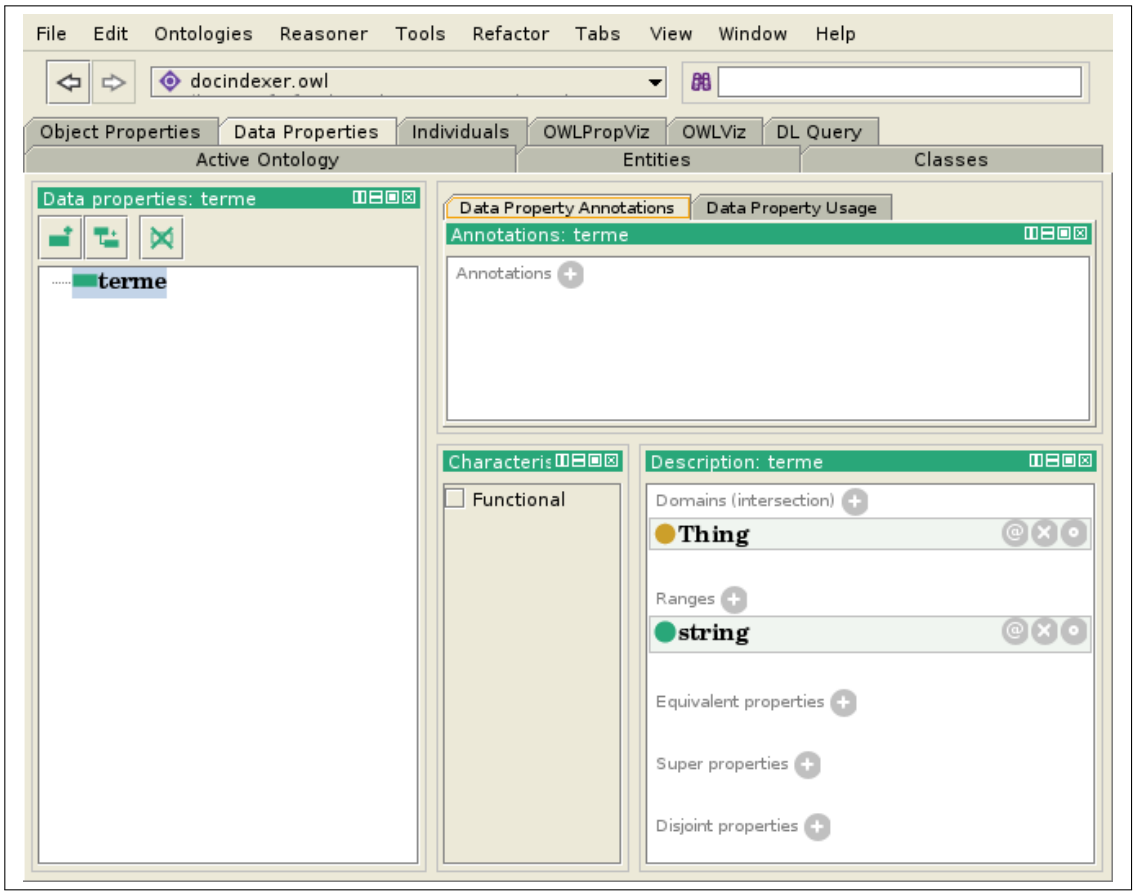

FIGURE 4.1 – Capture d'écran de l'outil « Protégé » montrant l'attribut « terme »

En recoupant tous ses paramètres, le domaine que je maîtrise le mieux avec un accès facile à de nombreux documents de qualité est le domaine de `Star Wars` et plus particulièrement le contenu du site « `Starwars-Holonet.com` » qui est l'une des plus grande encyclopédie en ligne francophone sur ce domaine.

Cependant, l'univers de `Star Wars` étant particulièrement riche, réalisé une *ontologie* générale de celui-ci est une tâche bien trop conséquente vis-à-vis de ce travail et nous nous limiterons donc à un petit sous-ensemble de cet univers de fiction.

Conformément à l'étape 1 de la méthodologie choisie pour la construction d'une *ontologie* (voir section 3.2.1), on doit commencer par poser une série de questions de base pour essayer de cerner plus correctement le domaine. En voici quelques unes :

1. À quel(s) groupement(s) et/ou organisation(s) appartient un vaisseau ?
2. Quels sont les différents termes employés pour parler de tel personnage ?
3. Quelle(s) espèce(s) peuple(nt) telle planète/lune/station ?
4. Qui est le chef de tel groupement/organisation ?

4.3.3 Cherche une ontologie existante

L'étape suivante dans le développement d'une *ontologie* est la recherche de ressources existantes qui nous éviteraient de devoir tout construire en partant de zéro. Ou même mieux, de trouver une *ontologie* existante qui nous permettrait de ne pas devoir la construire nous-mêmes.

La recherche d'une *ontologie* existante dans le domaine c'est avouée veine. Que ce soit par les bibliothèques d'*ontologies*, sur les moteurs de recherche documentaire ou en passant par la communauté `Star Wars` francophone.

Cependant, le site « `Starwars-Holonet.com` » fournit un classement de ses fiches encyclopédiques qui peut être repris, au moins partiellement, dans notre *ontologie*. Une capture d'écran de ce classement se trouve à la figure 4.2.

4.3.4 Construire sa propre ontologie

Maintenant qu'on a le domaine de l'*ontologie* et qu'on a fait le travail de recherche de ce qui existait pour ce domaine, il est temps de passer à la construction de l'*ontologie* à proprement parlé. Cela correspond aux étapes trois à sept de la méthodologie tirées de [Noy *et al.*, 2005] telles qu'elles sont expliquées à la section 3.2.3.

Il est à noter que l'outil utilisé pour la construction de l'*ontologie* s'appelle *Protégé*. Cet outil est le même que celui utilisé dans [Noy *et al.*, 2005]. Pour plus d'informations sur cet outil, consulter [pro, 2009].

TABLE 4.4 – Liste des termes importants du domaine de `Star Wars`

Humain	Protocolaire	Naboo	Leïa
droïd	Empire	République	Vador
planète	lune	vaisseau	espèce
organisation	affiliation	chef	Centerpoint
Palpatine	Exécutor	Aldéraan	Skywalker

Étape trois : lister les termes

La tâche suivante à réaliser est l'étape trois qui consiste à rédiger une liste de termes liés au domaine de notre *ontologie* et se mettre d'accord sur leur signification. Dans le domaine de `Star Wars`, un grand intérêt est attaché à bien catégorisé les sources d'information suivant qu'elles soient considérées comme officielles ou non. Et, lorsque deux sources officielles parlent d'un même terme, d'un même personnage, d'un même vaisseau, ... de manière différente, un point d'honneur est apporté par la communauté et par les responsables de `Star Wars` pour trouver un moyen de faire cohabiter les deux. La partie visant à se mettre d'accord sur la signification des termes n'est dès lors pas compliquée, il suffit de reprendre la signification officielle.

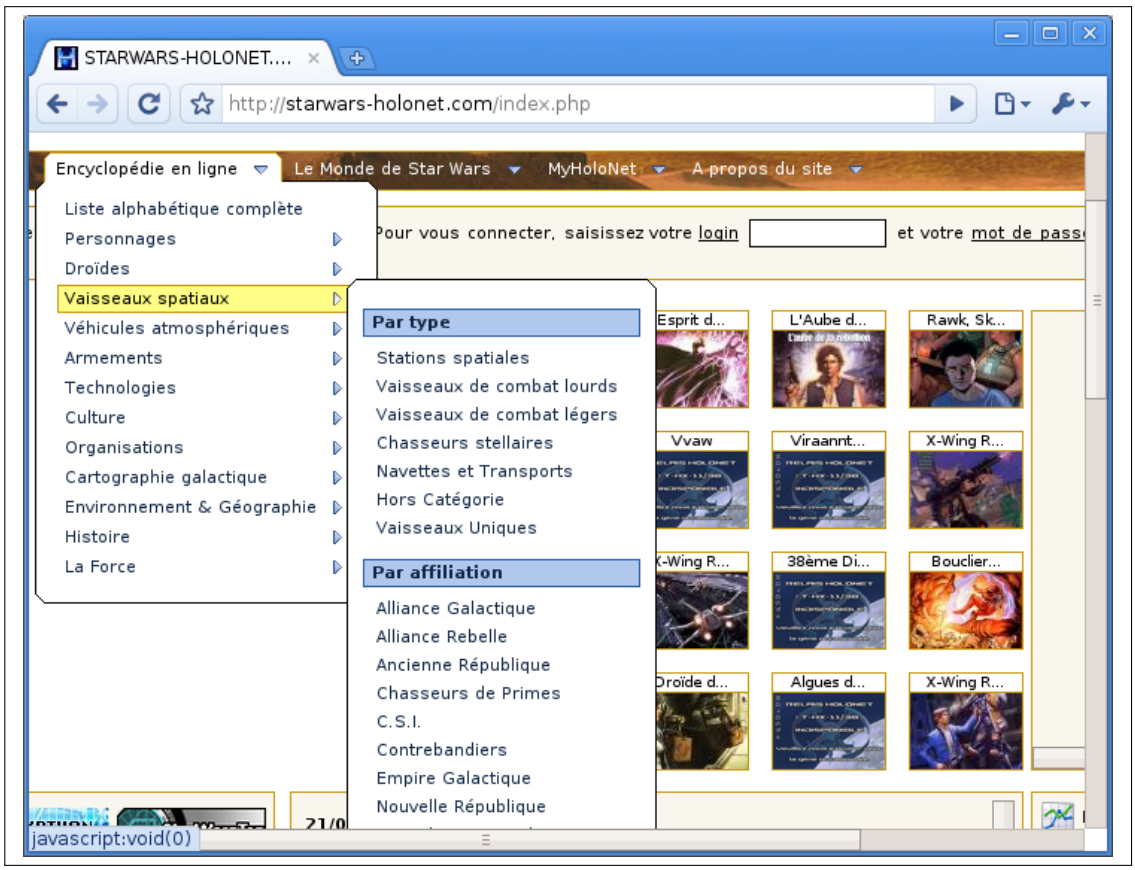

FIGURE 4.2 – Classements des fiches sur le site « `Starwars-Holonet.com` »

La liste des termes réalisée dans le cadre de se travail a été reproduite à la table 4.4 qui se trouve à la page 58. Cette liste est loin d'être exhaustive mais fournit une bonne base pour la suite de la construction.

Étape quatre : taxonomie

L'étape suivante consiste à extraire les concepts de la liste de la section précédente et à les organiser de manière hiérarchique pour former ce que l'on appelle une *taxonomie*. On obtient alors la figure 4.3.

L'approche suivie est l'approche « *top-down* ». On commence par définir les concepts de plus haut niveau qui sont ici :

Espèce Dans ce concept, on regroupe les races et espèces d'être vivants ainsi que les modèles de droïds[8].

Lieu L'ensemble des lieux « stellaire » où des personnes vivent. (planètes, lunes, astéroïdes, stations spatiales, ...)

Organisation Ce concept décrit les groupements de personnages. (Gouvernements galactiques, ...)

Personnage Ce concept regroupe tous les personnages.

Transport Ce concept regroupe les objets artificiels permettant de transporter des personnes ou du matériel. (Speeders, vaisseaux spatiaux, stations spatiales même si ces dernières ne se déplacent pas toutes, ...)

Une fois le premier niveau de concept terminé, on peut raffiner éventuellement ce qui vient d'être fait en introduisant des sous-concepts :

- Espèce :

  Droïd Les droïds sont des formes de vie artificielles. Les personnages qui sont des droïds appartiennent à des personnages qui sont organiques, la séparation de ce concept en suivant cette différence permettra donc de supporter la relation par la suite.

  Organique Ce concept reprend toutes les espèces organiques.

- Lieu :

  Naturel Ce concept regroupe les lunes, les planètes, les astéroïdes, ... tout ce qui n'a pas été créé artificiellement.

  Station spatiale Ici, c'est tout ce qui n'existe pas naturellement mais a été créé par une espèce. La séparation de ce concept de « Lieu » et de « Naturel » se fait car se concept hérite du concept de « Transport (spatial) » en plus de celui de « Lieu » ce qui n'est pas le cas de son concept-frère.

[8] un droïd est une sorte de robot

TABLE 4.5 – Liste des attributs de l'ontologie

#	**Nom**	**Domaine**
1	Terme	Thing
2	Nom	Thing
3	Prénom	Personnage
4	Type	Transport

- Transport :

  Station spatiale C'est le même concept que dans « Lieu ».

Étape cinq : les propriétés

L'étape suivante, la cinquième, consiste à enrichir les différents concepts de l'étape précédente avec des propriétés. Ces propriétés peuvent être des attributs (valeur simple) ou des relations (ayant pour valeur des instances de concepts).

Commençons par les attributs qui sont appelés ici *Data Properties*. Comme vu à la section 4.3.1, le premier attribut que l'on doit définir et le plus important est l'attribut « Terme » qui sera l'élément principal utilisé par la suite. Dans les autres attributs identifiés, il y a le « Nom » qui peut caractériser l'ensemble des différents concepts de la section précédente. Le « Prénom » qui lui ne caractérise que les « `Personnages` » et enfin le « Type » qui caractérise les « `Transports` ».

Après les attributs, c'est le tour des relations qui sont appelées, elles, *Object Properties* en *OWL*.

Une première relation — « `appartientA` » — exprime le fait qu'un « `Personnage` » appartient à une certaine « `Espèce` ».

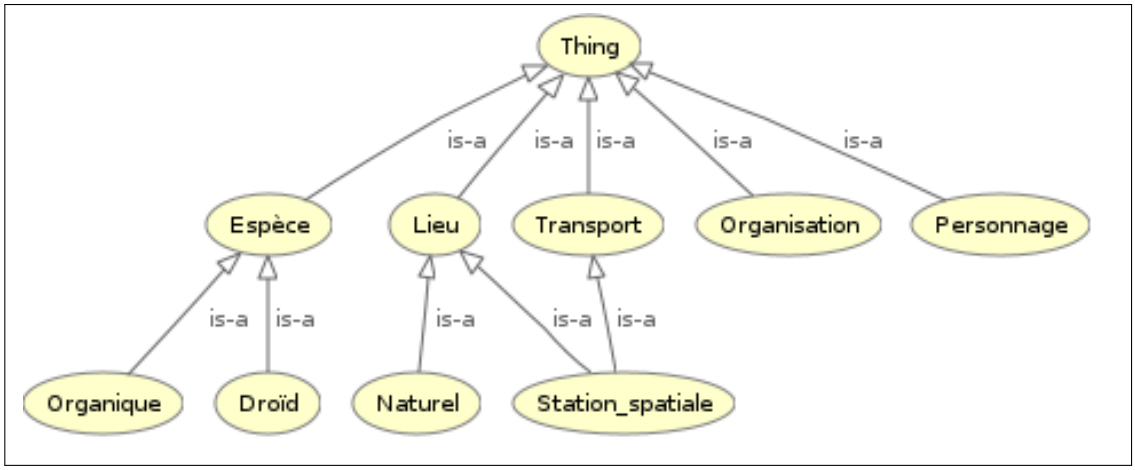

FIGURE 4.3 – Taxonomie de l'ontologie construite

TABLE 4.6 – Liste des relations de l'ontologie

#	**Nom**	**Domaine**	**Type**
1	appartientA	Personnage	Espèce
2	estLaPropriétéDe	appartientA *some* Droïd	appartientA *some* Organique
3	estMembreDe	Personnage	Organisation
4	aPourMembre	Organisation	Personnage
5	aPourMembreClé	Organisation	Personnage
6	estPossédéPar	Transport	Organisation
7	estOriginaireDe	Personnage	Lieu
8	aPourReprésentant	Lieu	Personnage

Une deuxième relation — « `estLaPropriétéDe` » — fixe le fait qu'un droïd est la propriété d'une personnage organique. Pour ce faire, on utilise l'inférence en mettant comme domaine (respectivement type) de la relation un concept qui n'existe pas physiquement mais l'ensemble de ce qui est lié à « `Droïd` » (respectivement « `Organique` ») par la relation « `appartientA` ». Vu la définition de la relation « `appartientA` », il s'agit bien de « `Personnage` » ce que *Protégé* détecte et affiche comme montré sur la figure 4.4 où les deux cadres avec des bordures en pointillés[9] n'ont pas été définis mais bien inférés par l'outil.

Une troisième relation — « `estMembreDe` » — met en avant le fait que les « `Personnages` » peuvent être membre d'« `Organisations` ».

La quatrième relation — « `aPourMembre` » — est la relation inverse de la troisième et relie donc les « `Organisations` » aux « `Personnages` ».

De même, une cinquième relation — « `aPourMembreClé` » — représente des membres importants et est donc un sous-ensemble de la quatrième relation. Ces trois relations sont très liées entre elles et l'inférence de l'*ontologie* va nous permettre de gérer cela très proprement. En effet, si on définit qu'un « `Personnage` » est un membre d'une « `Organisation` », alors la relation inverse disant que l'« `Organisation` » à ce « `Personnage` » pour membre est inférée directement par le système. De même, si on définit un « `Personnage` » comme étant un membre clé d'une « `Organisation` », la relation « `aPourMembre` » sera automatiquement considérée comme présente sans avoir besoin de la spécifier et la relation inverse « `estMembreDe` » également.

Ensuite une sixième relation — « `estPossédéPar` » — met en avant la propriété de possession d'un « `Transport` » par une « `Organisation` ».

Vient ensuite les relations sept et huit — respectivement « `estOriginaireDe` » et « `aPourReprésentant` » — qui mettent en avant le fait qu'un « `Personnage` » est originaire d'un certain « `Lieu` » et donc qu'un « `Lieu` » possède tout un ensemble de « `Personnages` » qui le représentent à travers la galaxie.

Par le biais de ces huits relations, on a relié entre eux tous les concepts de notre *ontologie*.

[9] Et présenté avec un fond jaune dans la version électronique

TABLE 4.7 – Mise à jour de la liste des attributs de l'ontologie du tableau 4.5 avec l'ajout de leurs facettes type, cardinalités minimales et maximales

#	Nom	Domaine	Type	Cardinalité
1	Terme	Thing	String	0..*
2	Nom	Thing	String	0..*
3	Prénom	Personnage	String	0..*
4	Type	Transport	String	1..1

L'outil *Protégé* ne permet pas d'afficher sous forme graphique les relations aussi facilement et lisiblement qu'il ne le permet pour la *taxonomie* ce qui explique l'absence de schéma pour cette étape. Les tableaux 4.5 et 4.6 reprennent respectivement l'ensemble des attributs et celui des relations.

Étape six : les facettes

L'étape six a pour but de préciser les facettes associées aux propriétés découvertes dans l'étape précédente. Dans cette étape, on spécifie les cardinalités et on définit plus précisément le domaine et le type.

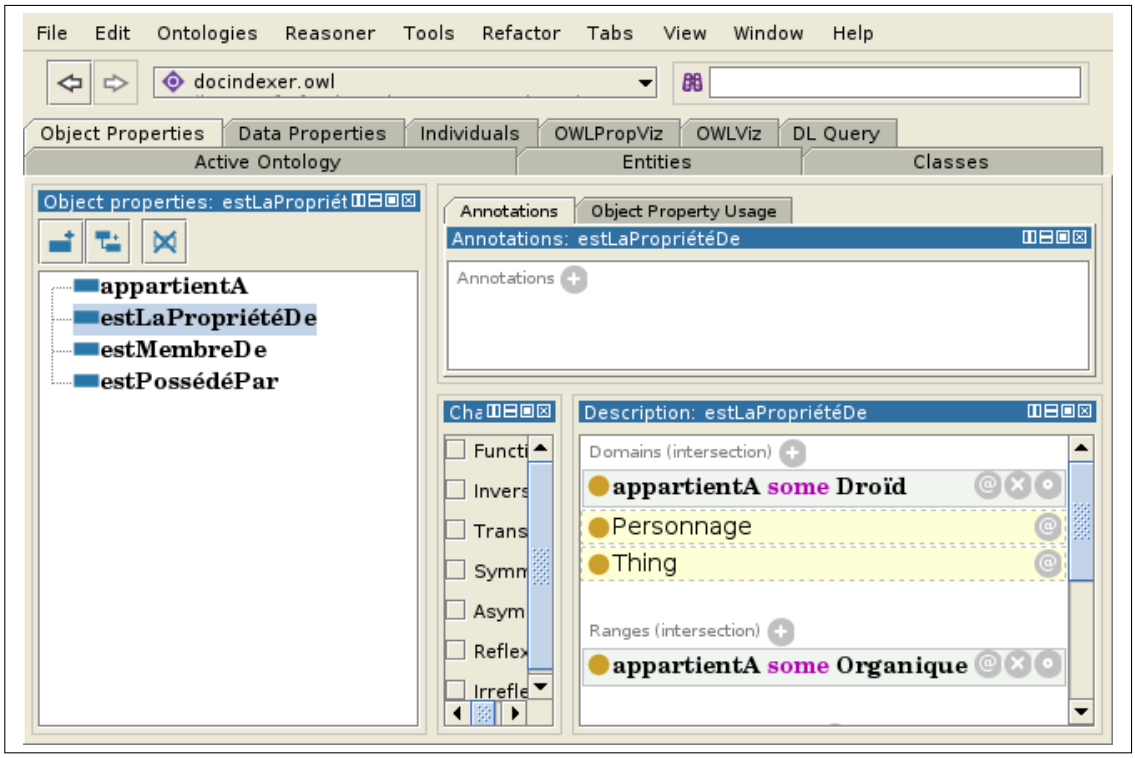

FIGURE 4.4 – Exemple d'inférence effectuée par l'outil Protégé sur notre ontologie de test

Reprenons les éléments dans le même ordre que lors de l'étape précédente et commençons donc par les attributs.

Le premier est l'attribut « `Terme` » qui a pour type une chaîne de caractères et prend un nombre quelconque et éventuellement nul de valeurs pour chaque concept.

L'attribut « `Nom` » dans l'univers `Star Wars` va de zéro valeur pour une chose ou une personne anonyme à un pour les cas normaux. Cependant il arrive que certaine personnes possèdent plusieurs noms. On ne fixera donc pas de limite supérieur pour cet attribut. En ce qui concerne le type, l'attribut prend pour valeur une chaîne de caractères.

Pour l'attribut « `Prénom` », c'est de nouveau une chaîne de caractères. Pour les valeurs, dans l'univers de `Star Wars`, on peut ne pas en avoir comme on peut en avoir plusieurs, on aura donc un nombre quelconque et éventuellement nul de valeurs.

Enfin, il y a l'attribut « `Type` » qui est de nouveau une chaîne de caractères et qui a forcement une et une seule valeur pour chaque vaisseau même si cette valeur peut être « inconnu ».

Ce qui nous permet de compléter le tableau 4.5 pour arriver au tableau 4.7.

Il nous reste maintenant à appliquer le même raisonnement pour les relations. Sauf que nous ne devons plus définir que les cardinalités ici.

La première relation, « `appartienA` » qui relie un « `Personnage` » à une « `Espèce` » a pour cardinalité minimale un. Il est impossible dans l'univers de `Star Wars` de ne pas appartenir à une « `Espèce` ». Pour la cardinalité maximale, on pourrait se contenter de la mettre à un également. Le croisement entre « `Espèce` » étant dans les faits inexistant. Cependant cela reste théoriquement possible, notamment le croisement entre deux types de « `Droïds` ».

La deuxième relation, « `estLaPropriétéDe` » relie les « `Personnages` » qui appartiennent à une instance du concept « `Droïd` » à des « `Personnages` » qui appartiennent eux à une instance du concept « `Organique` ». Dans l'univers de `Star Wars`, les « `Droïds` » appartiennent toujours à quelqu'un mais on ne connaît pas forcement cette personne et on mettra donc comme cardinalité minimale zéro. Pour la cardinalité maximale, rien n'oblige un « `Droïd` » à n'être la propriété que d'une personne et on ne mettra donc pas de limite sur la cardinalité maximale.

La troisième relation, « `estMembreDe` », montre le lien qui existe entre des « `Personnages` » et des « `Organisations` ». Nous accepterons pour cette relation qu'un « `Personnage` » soit relié à un nombre quelconque et éventuellement nul d'« `Organisations` ».

En ce qui concerne la quatrième relation, « `aPourMembre` », qui est en réalité l'inverse de la troisième relation, les cardinalités sont les mêmes. Une « `Organisation` » peut très bien ne pas[10] avoir de membres tout comme elle peut en avoir un très grand nombre.

La cinquième relation, « `aPourMembreClé` », est une « sous-relation[11] » de la précédente ce qui signifie que toute valeur de cette relation doit être présente dans sa « super-relation ». Donc, vu qu'on a définit la cardinalité minimale à zéro pour la quatrième relation ce qui se traduirait par une absence de membre, la cardinalité minimale de cette relation-ci doit être également de zéro. Si on a pas de membre, aucun d'entre eux ne

[10] Ce qui correspondrait probablement à une « `Organisation` » qui n'existe plus

[11] L'équivalent d'une sous-classe ou d'un sous-concept porté au niveau des relations

TABLE 4.8 – Mise à jour de la liste des relations de l'ontologie du tableau 4.6 avec l'ajout facettes décrivant leurs cardinalités minimales et maximales

#	Nom	Domaine	Type	Cardinalité
1	appartientA	Personnage	Espèce	1..1
2	estLaPropriétéDe	appartientA *some* Droïd	appartientA *some* Organique	1..*
3	estMembreDe	Personnage	Organisation	0..*
4	aPourMembre	Organisation	Personnage	0..*
5	aPourMembreClé	Organisation	Personnage	0..*
6	estPossédéPar	Transport	Organisation	0..*
7	estOriginaireDe	Personnage	Lieu	0..1
8	aPourReprésentant	Lieu	Personnage	0..*

peut être un membre clé. Pour la cardinalité maximale, c'est le même principe. Tous les membres peuvent être des membres clés et donc la cardinalité maximale est non bornée ici aussi.

Ensuite, la sixième relation, « `estPossédéPar` », permet de relier les « `Transports` » et les « `Organisations` ». Pour des petits transports, ils peuvent appartenir à un « `Personnage` »[12] directement, ce qui n'est pas représenté dans notre *ontologie* et n'appartiennent donc pas à une « `Organisation` » et on a donc une cardinalité minimale de zéro. Pour la cardinalité maximale, il est envisageable qu'un transport appartiennent à plusieurs « `Organisations` » et on n'a donc pas de limite supérieure pour la cardinalité maximale de cette relation.

La septième relation, « `estOriginaireDe` », relie un « `Personnage` » avec le « `Lieu` » d'où il est originaire. On Peut ne pas connaître son origine et donc la cardinalité minimale est de zéro. Ensuite, on ne peut pas être né à plusieurs endroits différents et donc la cardinalité maximale est de un.

La huitième et dernière relation, « `aPourReprésentant` », est la relation inverse de la septième. Elle relie un « `Lieu` » à l'ensemble des « `Personnages` » qui en sont originaires. Il y a donc un nombre quelconque et éventuellement nul de représentants d'un « `Lieu` ».

Tout ceci nous permet, comme cela a été fait pour les attributs, de mettre à jour le tableau 4.6 réalisé pour l'étape précédente. Et nous obtenons dès lors le tableau 4.8.

Étape sept : les instances

La septième et dernière étape consiste à créer des instances pour peupler notre *ontologie*. Pour commencer, le plus simple consiste à reprendre les mots de la liste de l'étape trois tel que repris dans le tableau 4.4 en tenant compte toutefois que la majorité d'entre eux ont été utilisé au niveau des concepts, des attributs et des relations.

[12] On peut même aller plus loin et dire qu'il s'agit d'un « `Personnage` » appartenant à une « `Espèce` » « `Organique` »

Reprendre l'intégralité des instances une par une en les expliquant n'aurait pas un grand intérêt pour le travail aussi nous nous contenterons d'en détailler une : *Leïa Organa Solo* en commençant par préciser qu'une capture d'écran de la définition de cette instance se trouve à la figure 4.6. Un tableau reprend l'ensemble des instances. Il s'agit du tableau 4.9. La figure 4.5 montre l'*ontologie* finale avec toutes les instances. Pour éviter d'avoir un schéma trop chargé, les relations sont montrées uniquement entre les instances et seulement celles nécessaires (celles qui peuvent être inférées comme les relations inverses ne sont pas représentées). De même les attributs ne sont pas présents.

Leïa est un personnage important dans `Star Wars`, elle appartient à la haute sphère de la Nouvelle République. On peut donc considérer que c'est un personnage important de cette organisation. On va donc commencer par créer la relation « `aPourMembreClé` » entre l'instance « `République`[13] » et l'instance « `Leïa` ». De ce fait, le moteur d'inférence ajoute qu'il sait que l'instance « `Leïa` » est un « `Personnage` » et qu'il existe une relation « `estMembreDe` » entre elle et l'instance « `République` » et une relation « `aPourMembre` » dans l'autre sens. Et voila la majorité des informations créées avec une simple relation.

Ensuite, on peut spécifier la relation « `estOriginaireDe` » entre notre instance « `Leïa` » et l'instance « `Aldéraan`[14] ». Et le moteur d'inférence en déduit directement qu'il existe une relation « `aPourReprésentant` » entre « `Aldéraan` » et « `Leïa` ».

Enfin, spécifions une dernière relation, « `appartientA` » pour spécifier que « `Leïa` » est lié à l'instance « `Humain` » qui est bien entendu une instance de « `Espèce` » (comme le déduira le moteur d'inférence de l'*ontologie*) mais plus précisément une instance d'« `(Espèce) Organique` ».

Une fois les relations spécifiées, passons aux attributs. *Leïa Organa Solo* a un prénom : *Leïa* et deux noms : *Organa* et *Solo*. Pour les termes, on peut reprendre *Leïa* vu qu'elle est la seule à porter ce prénom et *Organa*. Par contre, *Solo* nous donnerait beaucoup trop de mauvais résultats vu le nombre important d'autres personnes portant ce nom.

4.4 Association indexeur et ontologie

L'*ontologie* seule n'a pas grand intérêt dans ce travail. Il faut la coupler avec l'indexeur pour pouvoir en tirer un profit dans le moteur de recherche documentaire par la suite. Le but de cette section est d'expliquer comment le lien a été réalisé entre les deux précédents outils développés.

4.4.1 Utilisation d'une ontologie en java

Pour pouvoir faire travailler ensemble l'*ontologie* au format *OWL* avec l'indexeur réalisé en *Java*, il faut mettre en place un moyen de lire et comprendre l'*ontologie*

[13] l'instance « `République` » devient dès lors obligatoirement une instance du concept « `Organisation` »

[14] l'instance « `Aldéraan` » devient dès lors obligatoirement une instance du concept « `Lieu` »

en *Java*. Cela a été fait dans le cadre de ce travail par l'utilisation d'un *framework* nommé *Jena* et d'un *reasoner* nommé *Pellet* qui s'intègre très bien avec *Jena*[15].

[15]Plus d'information sur *Jena* sur le site `Web` : [jen, 2009]

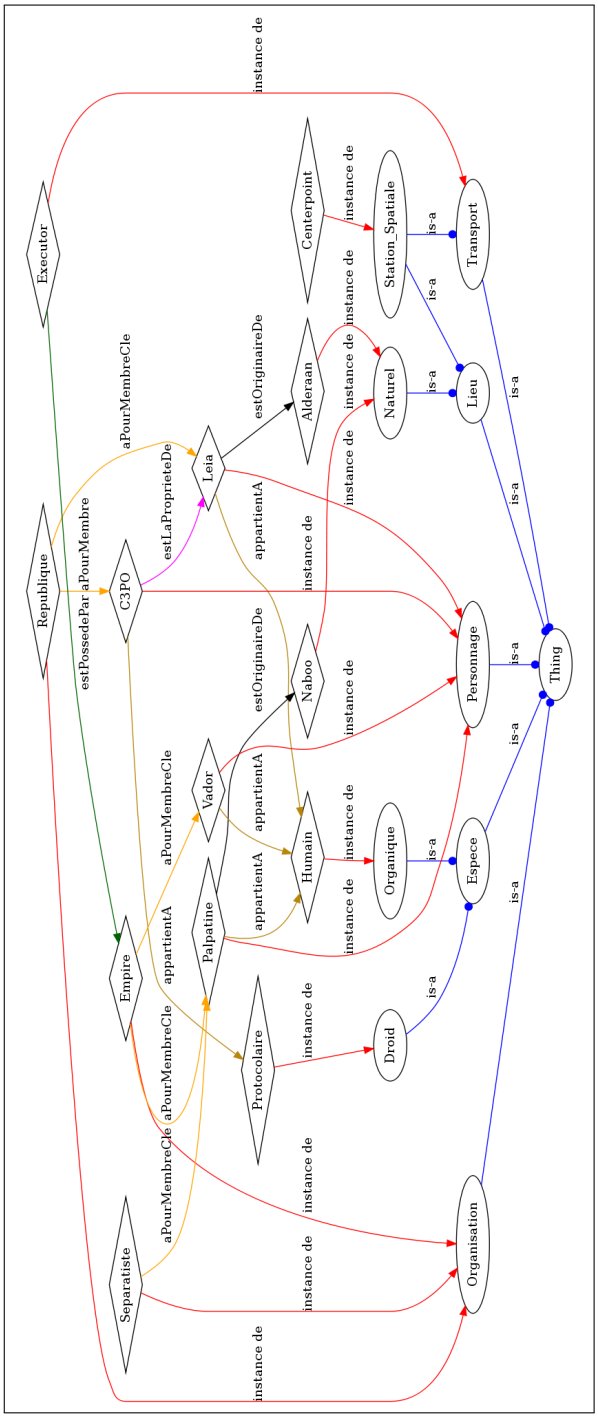

FIGURE 4.5 – Représentation graphique de l'ontologie terminée

En utilisant le *framework Jena* seul, il faudrait faire les raisonnements « à la main » ce qui prendrait beaucoup de temps, ne serait pas efficace du tout et ferait grimper le risque d'erreur. *Pellet* apporte avec lui un mécanisme d'interrogation très avancé : *SPARQL Protcol and RDF Query Language (SPARQL)*[16] qui, comme son nom l'indique, est directement inspiré de *Structured query language (SQL)*.

4.4.2 Principe de couplage

Le couplage entre l'indexeur et l'*ontologie* est assez basique dans ce travail. Lors de l'analyse d'un document dans l'indexeur (voir section 4.2.4 page 54) on va regarder terme

[16] *SPARQL* n'est pas abordé en détail dans le présent travail mais plus d'informations peuvent être trouvées dans [Prud'Hommeaux *et al.*, 2006].

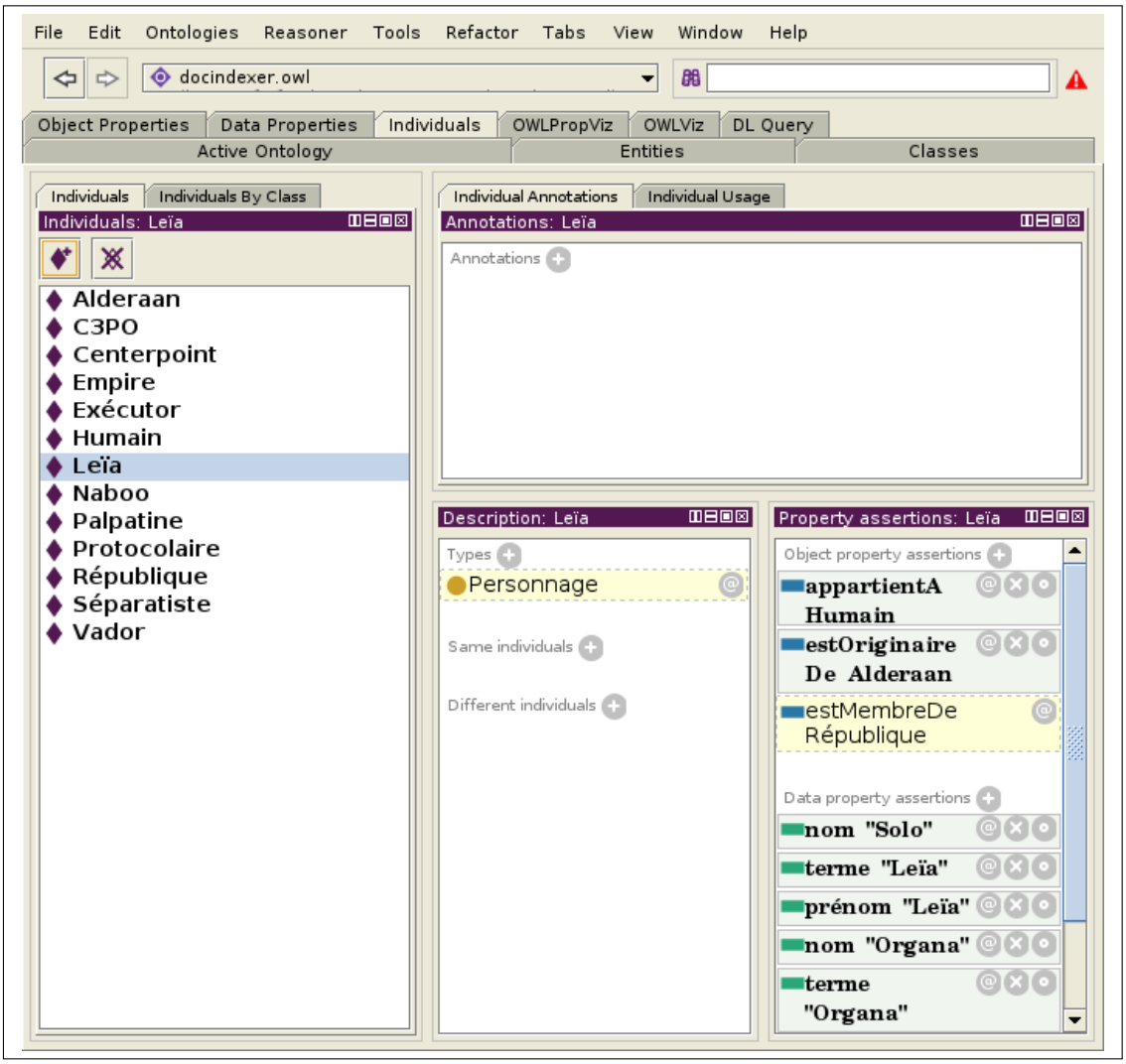

FIGURE 4.6 – Capture d'écran de la définition de l'instance représentant *Leïa Organa Solo* dans l'outil Protégé

TABLE 4.9 – La liste des instances de l'ontologie `Star Wars`

#	**Nom**	**Concept**
1	Leïa Organa Solo	Personnage
2	Dark Vador	Personnage
3	Palpatine	Personnage
4	C3PO	Personnage
5	Humain	(Espèce) Organique
6	Protocolaire	(Espèce) Droïd
7	Naboo	(Lieu) Naturel
8	Aldéraan	(Lieu) Naturel
9	Centerpoint	(Lieu)(Transport)Station_spatiale
10	Empire	Organisation
11	(Nouvelle) République	Organisation
12	Corporation des Systèmes Indépendants	Organisation
13	Exécutor	Transport

Listing 4.1 – Requête faisant le lien entre l'ontologie et l'indexeur

```
1 PREFIX docindexer: <http://info.fundp.ac.be/~notjacqu/docindexer.owl#
    >
2 SELECT DISTINCT ?instance ?terme
3 WHERE {
4         ?instance docindexer:terme ?terme;
5                   docindexer:terme ?searched.
6                   FILTER regex(str(?searched), "separatiste", "i")
7          }
```

par terme[17] le contenu du document et, pour chaque terme, on regarde dans l'*ontologie* si ce terme est rattaché à un concept via la relation « terme » justement.

Cette recherche se fait à l'aide de la requête montrée dans le listing 4.1 de la page 68. La syntaxe étant très proche de celle de *SQL*, il n'est pas très compliqué d'expliquer la requête :

1. On demande des instances et des termes présents dans l'*ontologie*

2. On spécifie que les termes en questions doivent être liés aux instances via l'attribut[18] « terme ».

[17] cette lecture terme par terme se fait très facilement à l'aide de mécanismes mis en place à l'aide de la librairie *Lucene*

[18] En regardant la ligne 4 uniquement, on ne peut pas dire s'il s'agit d'un attribut ou d'une relation

3. La ligne suivante (la ligne 5) dit qu'il doit exister une valeur de l'attribut « terme » pour les instances en question que l'on baptise « searched ». Ce nom est défini dans la requête comme l'on définirait un alias en *SQL*.

4. Enfin, la dernière ligne dit que « searched » est une chaîne de caractères et que celle-ci ne peut avoir une valeur qui ne correspondrait pas à la *RegExp* passée comme deuxième argument[19].

En résumé, on spécifie une *RegExp* qui va capturer un mot présent dans le texte, on va récupérer toutes les instances qui possèdent un terme qui répond positivement à cette *RegExp* et on va récupérer l'ensemble des termes associés à ces instances.

Pour chaque mot d'un document, on a donc une liste éventuellement vide de termes associés à celui-ci dans l'*ontologie*. On peut donc ajouter un nouveau champ dans l'*index* avec l'ensemble de ses termes liés à l'*ontologie* et pas forcement présents dans le contenu. Ainsi, on pourra faire une recherche sur un terme absent d'un document mais attaché à une instance qui possède un autre terme présent dans ce document et le considérer comme un document pertinent.

L'utilisation faite de l'*ontologie* dans ce travail est assez basique dans le sens où elle ne fait pas vraiment appel à l'inférence et ne tire donc pas avantage de l'*ontologie* autant qu'elle le pourrait. Cependant, l'utilisation qui en est faite devrait déjà permettre de voir une amélioration dans le moteur de recherche. Avec plus de temps, nous pourrions travailler sur une meilleur exploitation de l'*ontologie* en utilisant, par exemple, les relations de l'*ontologie*. Actuellement, on arrive à identifier une instance lié à la requête de l'utilisateur. Si on pouvait suivre des relations partant de cette instance vers d'autres et qu'on arrivait à identifié certaines des ces relations comme importantes pour la requête, on pourrait grandement améliorer les résultats. Cependant, cette tâche demande plus de temps et n'a donc pas été abordée dans ce travail.

4.5 Moteur de recherche documentaire

4.5.1 Introduction

Finalement on en arrive à la dernière brique de développement, le moteur de recherche documentaire qui va nous permettre d'exploiter tout ce qu'on a fait jusqu'ici.

Une interface simple et fonctionnelle pour un moteur de recherche est l'utilisation d'une page `Web`. L'*index* étant au format *Lucene*, il serait préférable d'utiliser un langage pourvu d'une librairie pour ce format histoire de ne pas devoir l'implémenter nous-mêmes. Les deux choix de langages les plus naturels sont dès lors *PHP* qui possède un portage de la librairie *Lucene* via le *framework* `Zend` d'un côté et *Java* de l'autre via des *JaveServer Pages (JSPs)* ou des *servlets*. Vu que tout le reste de l'application est en *Java*, le choix se porte donc assez naturellement pour continuer en ce sens.

[19] le troisième argument, le « i », précise que la *RegExp* est insensible à la casse

4.5.2 Interface de recherche

Pour rappel, cet outil doit être capable de rechercher des documents dans l'*index* sur base d'une série de mots-clés[20] fournie par un utilisateur. On doit pouvoir choisir de n'utiliser que l'*index* au sens traditionnel du terme ou de faire usage de l'*ontologie* pour enrichir le résultat.

De plus, on doit pouvoir spécifier le nombre de résultat que l'on souhaite obtenir pour pouvoir calculer précisément les indicateurs dans le chapitre 5. On obtient donc l'interface de l'image 4.7.

4.5.3 Interface d'affichage des résultats

Une fois la requête effectuée, il ne reste qu'à afficher les résultats. Le choix qui a été fait dans cet outil réside dans la création d'un tableau. Chaque ligne correspond à un document. Les quatres colonnes de ce tableau correspondent respectivement au rang du

[20] Ces mots-clés peuvent être enrichis à l'aide d'expressions booléennes comme des négations, des « ET » et/ou des « OU »

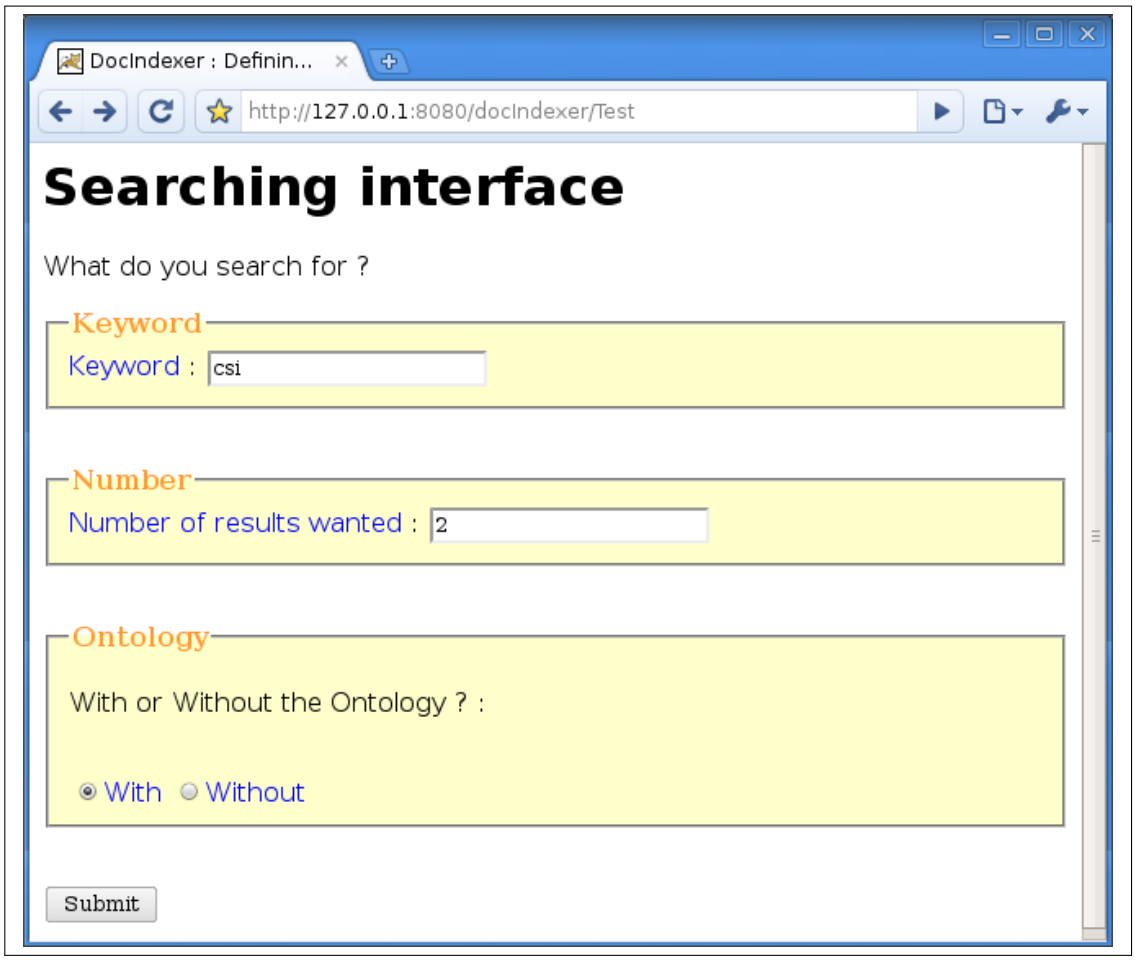

FIGURE 4.7 – Capture d'écran de l'interface de recherche

document dans les résultats, à son *URL*, à son titre (éventuellement vide) et à son score calculé par *Lucene* sur base de l'indicateur *TF-IDF*.

Les figures 4.8 et 4.9 sont des captures d'écran de l'interface de consultation des résultats. La première recherche sans utiliser l'*ontologie* alors que la deuxième effectue la même recherche en se basant, elle, sur l'*ontologie*.

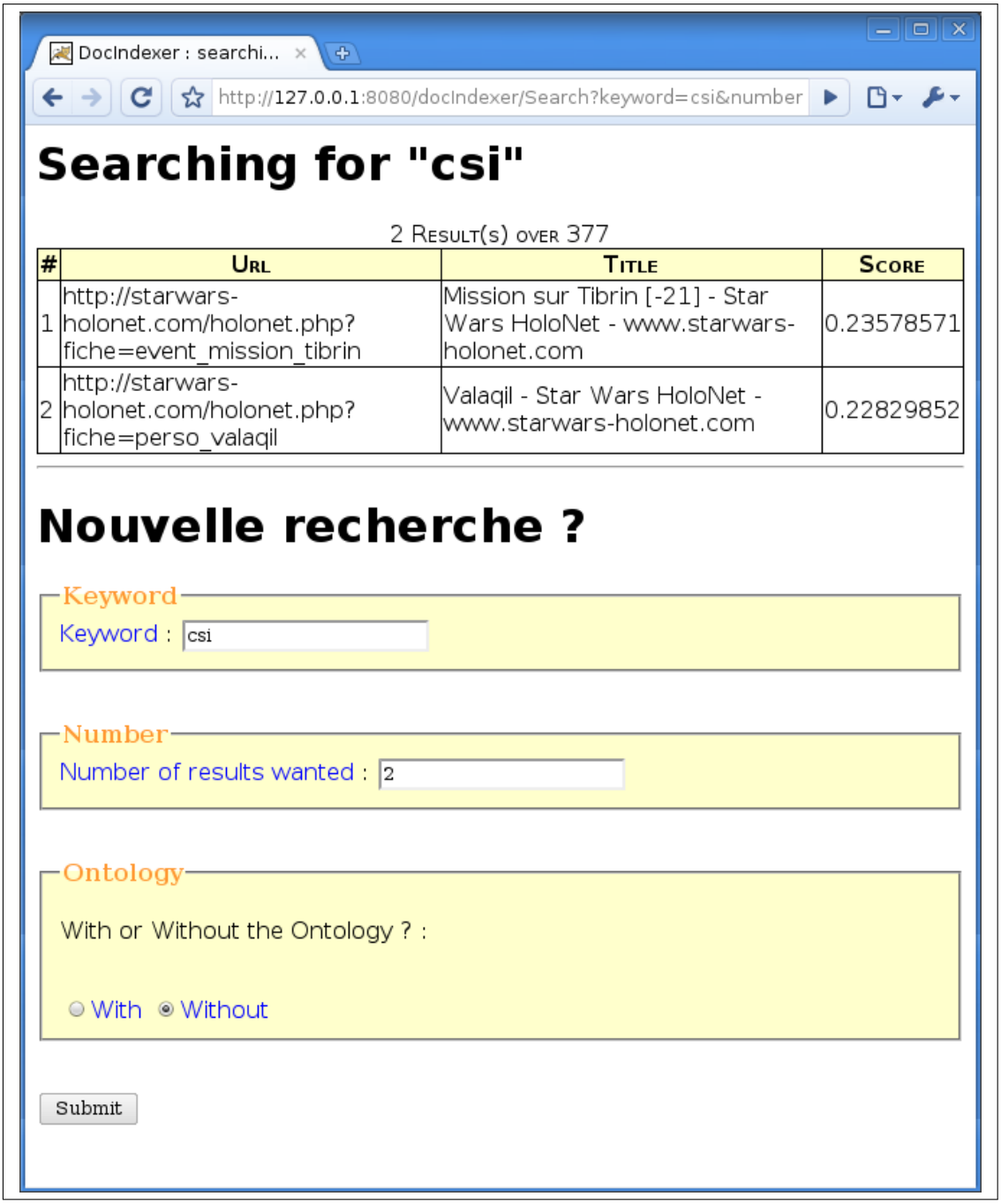

Figure 4.8 – Capture d'écran de l'interface de consultation des résultats sans l'ontologie

La différence entre l'utilisation ou non de l'*ontologie* joue sur les champs de l'*index* qui seront ou non utilisés par le moteur de recherche documentaire. Deux champs sont exploitables, le champs « Contenu » et le champ « Ontologie ».

Le premier, « Contenu », contient les termes des documents qui ont été indexés. Ce champ correspond donc au système « classique ». Et si on désire faire une recherche sans utiliser l'*ontologie*, on recherche les termes présents uniquement dans ce champ.

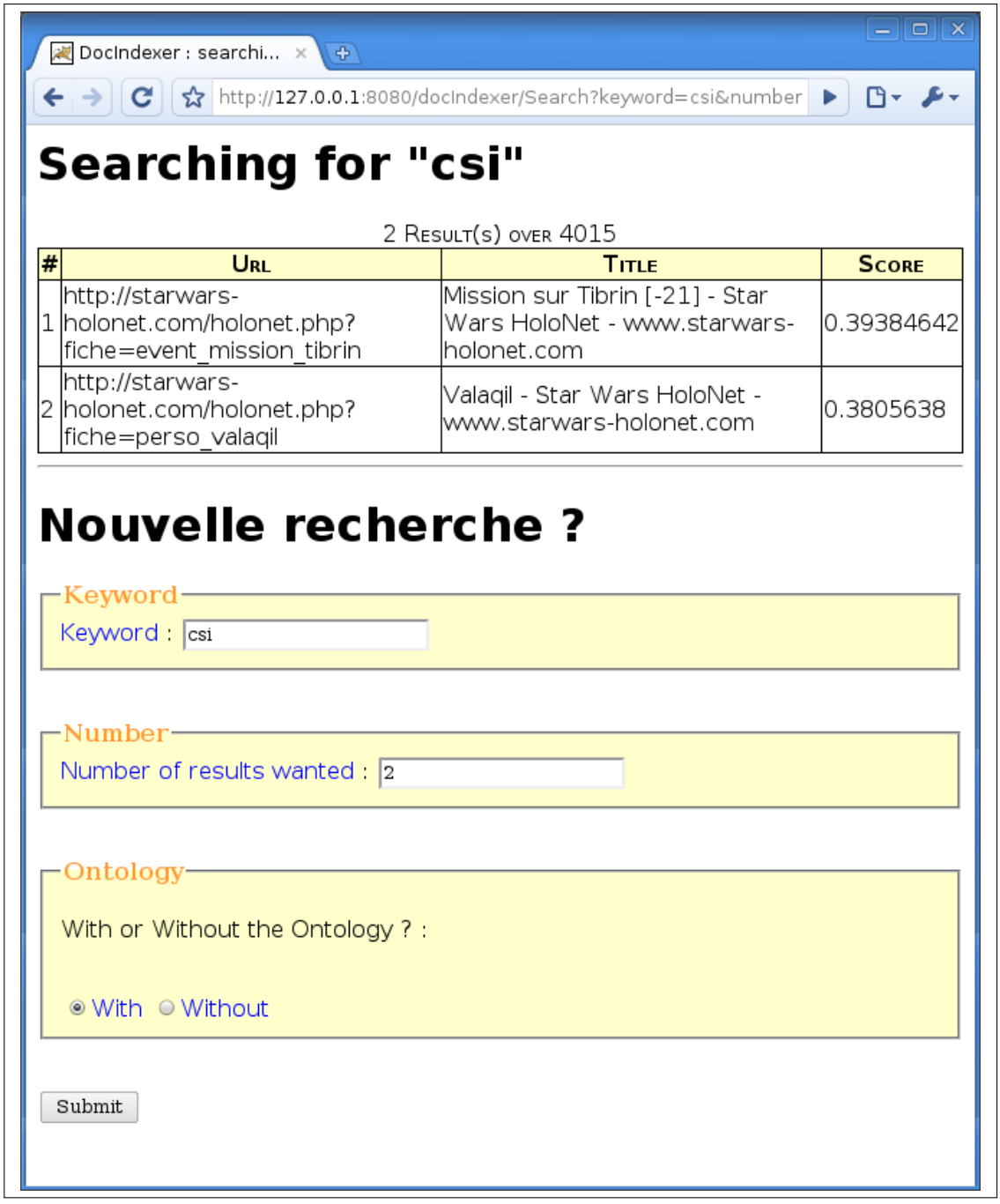

Figure 4.9 – Capture d'écran de l'interface de consultation des résultats avec l'ontologie

Le second champ, « Ontologie », contient les termes qui ont été extraits depuis l'*ontologie* comme expliqué à la section 4.4.2. Ce sont donc tous des termes présents dans l'*ontologie*. Si on recherche dans le moteur de recherche avec l'*ontologie* activée, il va regarder dans les deux champs. De cette façon, si on recherche un terme présent dans l'*ontologie*, on le trouvera dans le second champ. Et si le terme n'est pas présent dans l'*ontologie*, on retombe sur le système classique. Si on ne recherchait que sur le second champ, le fait de cherche un terme absent de l'*ontologie* se traduirait par une absence de résultat. L'utilisation des deux champs permet donc d'éviter ce scénario. Cependant, l'utilisation des deux champs induit un autre comportement. Supposons deux termes liés à une même instance, l'un présent dans le document, l'autre non. Si l'on se contente de chercher dans le second champ, chercher l'un ou l'autre n'a aucune influence sur les résultats retournés ou sur l'ordre dans lequel ils sont retournés vu que si l'un des deux est présents dans le texte, l'autre sera présent autant de fois par le mécanisme expliqué à la section 4.4.2. Cependant, vu que l'on recherche sur les deux champs, le terme réellement présent dans le document aura un poids plus important que l'autre et à partir de là, chercher un terme ou l'autre aura potentiellement un impact sur les résultats et sur l'ordre dans lequel ils apparaissent.

Chapitre 5

Étude de cas

5.1 Introduction

Dans ce chapitre, le but est d'évaluer ce qui a été réalisé pour voir si le moteur de recherche documentaire que l'on a construit permet d'obtenir de meilleurs résultats.

La première étape de cette évaluation, à la section 5.2, met en avant la « base » de test. Cela correspond au corpus de documents utilisé pour les tests, les requêtes qui seront soumises dans chaque configuration et enfin les métriques qui seront récupérées pour chaque exécution.

La deuxième étape, section 5.3, consiste à exécuter chaque requête sur le moteur de recherche documentaire réalisé au chapitre précédent sans utiliser l'*ontologie*.

La troisième étape, section 5.4, consiste à exécuter ces mêmes requêtes sur le même moteur de recherche documentaire mais en utilisant l'*ontologie* cette fois-ci.

La quatrième étape, section 5.5, consiste à exécuter toujours les mêmes requêtes mais cette fois-ci sur deux moteurs de recherches existants. À savoir celui de `Google`[1] et celui interne de « `Starwars-Holonet.com` ».

Enfin, la cinquième et dernière étape de cette évaluation se trouve à la section 5.6. Elle a pour but de mettre en avant l'analyse des métriques obtenues dans les autres sections de ce chapitre.

5.2 « Base » de test

Le but de cette section est de mettre en avant les conditions des tests visant à évaluer le moteur de recherche documentaire réalisé au chapitre précédent.

[1] en prenant soin de le restreindre au site « `Starwars-Holonet.com` »

5.2.1 Corpus de test

Le corpus de test utilisé pour ces évaluations est le même pour tous les systèmes. Il s'agit des fiches de l'encyclopédie en ligne « `Starwars-Holonet.com` ».

Il est à noter que certaines fiches bien que n'existant pas encore possèdent déjà une *URL* et les liens sont déjà présents dans les autres fiches vers celles qui n'existent pas encore. Notre indexeur a donc atterri sur ses fiches pré-créées ou plus précisément sur la page disant que la fiche est prévue mais pas encore disponible. Le problème est que cette page affiche un résumé et un lien vers deux autres fiches existantes prises au hasard ce qui va donc perturber les résultats des recherches mais les mécanismes de filtrage disponibles dans l'outil ne permettent pas de filtrer ces pages là.

5.2.2 Les requêtes de test

Le plus pertinent pour tester l'outil est de prendre des termes présents dans l'*ontologie*. Partons sur une base de cinq requêtes et prenons donc quatre termes de l'*ontologie* liés à des concepts différents et un cinquième lié à un même concept qu'un des quatre premiers termes :

1. Vador
2. Naboo
3. Palpatine
4. Sidious (autre nom de Palpatine)
5. csi

Ce qui nous donne deux « `Personnages` », un « `(Lieu) Naturel` » et une « `Organisation` ».

5.2.3 Les métriques

Les métriques utilisées dans ce chapitre sont basées sur les métriques présentées à la section 2.4.4 et rappelées à la section 3.5.

On exécutera chaque requête sur chacun des quatre[2] moteurs de recherche documentaire. En gardant les dix premiers résultats. Ce qui nous donnera quatre ensembles de dix documents.

[2]Pour simplifier la rédaction, on supposera que le moteur de recherche créé au chapitre précédent utilisé sans l'*ontologie* et avec celle-ci comme deux moteurs différents

TABLE 5.1 – Le nombre de documents pertinents pour chaque requête dans l'ensemble des documents retournés par au moins un des moteurs de recherche

#	**Requête**	**Valeur**
1	Vador	17
2	Naboo	
3	Palpatine	18
4	Sidious	18
5	csi	

$$E : L'ensemble\ des\ documents\ du\ corpus \tag{5.1}$$
$$P : fct\ I \rightarrow O | I \subseteq E \cap O = \{d \in I : d\ est\ pertinent\} \tag{5.2}$$
$$R_{i,j} : l'ensemble\ des\ résultats\ pour\ la\ requête\ i\ par\ le\ moteur\ j \tag{5.3}$$
$$F_j : \bigcup_{i=1}^{4} R_{i,j} \tag{5.4}$$
$$Precision_{i,j} : \frac{\#P(R_{i,j})}{\#R_{i,j}} \tag{5.5}$$
$$Rappel_{i,j} : \frac{\#P(R_{i,j})}{\#P(F_j)} \tag{5.6}$$

La première chose à faire ensuite est de définir quels documents sont pertinents et lesquels ne le sont pas. Comme expliqué à la section3.5, nous n'avons pas de liste avec cette information et nous avons fait le choix de ne pas créer cette liste pour tout le corpus. Nous allons donc prendre l'union des documents retournés par les différents moteurs de recherche documentaire (comme indiqué à l'équation 5.4) et ne définir les documents pertinents (et donc les non-pertinents par élimination) que pour ceux-ci.

À partir de là, on peut calculer la précision de la même manière qu'à la section 2.4.4 ce qui est illustré par l'équation 5.5.

Pour le rappel, la formule change un peu vu qu'on ne connaît pas le nombre exact de documents pertinents pour tout le corpus mais seulement celui d'un sous-ensemble représentant l'union des résultats. On va donc ajuster le rappel pour se baser sur cette donnée. Étant donné qu'on utilise la même valeur pour tous les moteurs de recherche, cela permet toujours de les comparer entre eux. L'équation 5.6 montre la formule du rappel tel que nous venons de le définir.

Le rappel étant défini, il faut maintenant compter le nombre de documents pertinents pour chaque requête. C'est ce qui est fait au tableau 5.1.

TABLE 5.2 – Valeurs en pourcent de l'indicateur « précision » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur classique

Requête	1	2	3	4	5	6	7	8	9	10
Vador	100%	100%	100%	100%	80%	83%	71%	75%	78%	80%
Naboo	100%	100%	67%	75%	80%	83%	86%	75%	67%	70%
Palpatine	100%	100%	100%	100%	80%	67%	71%	63%	56%	60%
Sidious	0%	0%	33%	25%	40%	50%	43%	38%	33%	30%
csi	100%	50%	33%	50%	60%	50%	43%	38%	33%	30%
Moyenne	80%	70%	67%	70%	68%	67%	63%	58%	53%	54%

5.3 Moteur de recherche documentaire « classique »

Le premier moteur de recherche documentaire est celui développé au chapitre précédent et configuré pour ne pas utiliser l'*ontologie*.

Pour la première requête, celle concernant « Vador », seuls les résultats cinq et sept ne sont pas pertinents ce qui nous donne pour les dix documents une précision de 80% et un rappel de 47%.

Pour la deuxième requête, qui concerne, elle, la planète « Naboo ». Il existe treize documents pertinents pour cette requête. Le système classique en retourne sept ce qui nous donne une précision de 70% et un rappel de 54%.

Pour la troisième requête, concernant le personnage de « Palpatine », il y a en tout dix-huit documents pertinents. L'exécution de ce moteur de recherche retourne six résultats pertinents (les résultats un, deux, trois, quatre, sept et dix). Ce qui nous donne une précision de 60% et un rappel de 33%.

Pour la quatrième requête, il s'agit d'un autre nom du même personnage, « Sidious ». Il y a donc également dix-huit documents pertinents à trouver. L'exécution n'en retourne plus que trois (les résultats trois, cinq et six) soit la moitié de la requête précédente. Cela fait chuter la précision à 30% et le rappel à 17%.

La cinquième requête concerne l'organisation des « csi ». Il existe quinze documents pertinents pour cette requête. Le système classique nous en sort seulement trois ce qui donne une précision de 30% et un rappel de 20%.

En faisant la moyenne des indicateurs pour les cinq requêtes, on obtient 54% pour la précision moyenne et 34% pour le rappel.

Deux tableaux récapitulatifs, le 5.2 et le 5.3 reprennent respectivement les différentes valeurs l'indicateur de précision et de l'indicateur de rappel pour chaque requête et chaque nombre de documents.

TABLE 5.3 – Valeurs en pourcent de l'indicateur « rappel » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur classique

Requête	1	2	3	4	5	6	7	8	9	10
Vador	6%	12%	18%	24%	24%	29%	29%	35%	41%	47%
Naboo	8%	15%	15%	23%	31%	38%	46%	46%	46%	54%
Palpatine	6%	11%	17%	22%	22%	22%	28%	28%	28%	33%
Sidious	0%	0%	6%	6%	11%	17%	17%	17%	17%	17%
csi	7%	7%	7%	13%	20%	20%	20%	20%	20%	20%
Moyenne	5%	9%	12%	18%	22%	25%	28%	29%	30%	34%

5.4 Moteur de recherche documentaire couplé à l'ontologie

Le deuxième système a passé à l'évaluation est le même moteur de recherche que la section précédente, c'est à dire, celui que nous avons construit mais cette fois paramétré pour utiliser l'*ontologie*.

Les requêtes sont les mêmes et soumises dans le même ordre. Nous commençons donc par la requête « Vador ». Cette fois-ci, il y a sept résultats positifs soit un de moins que pour le précédent moteur. Ce qui nous donne une précision de 70% et un rappel de 41%.

Vient ensuite la requête sur « Naboo » qui retourne six résultats pertinents soit toujours un de moins que le moteur précédent. La précision est ici de 60% et le rappel de 46%.

La troisième requête est celle sur « Palpatine » qui retourne six résultats soit autant que le moteur précédent. La précision et le rappel sont donc les mêmes que ceux du moteur précédent pour cette requête à savoir respectivement 60% et 33%.

La quatrième requête est celle sur « Sidious » qui est liée au même concept que celui de la requête précédente. Ce moteur retourne cinq résultats positifs soit un de moins que pour « Palpatine » mais deux de mieux que le résultat de cette requête avec le moteur précédent. La précision est ici de 50% et le rappel de 28%.

La cinquième requête recherche l'organisation « csi ». Ce moteur retourne quatre documents positifs ce qui est mieux que le moteur précédent et donne une précision de 40% et un rappel de 27%.

En faisant la moyenne sur l'ensemble des requêtes, on obtient 56% pour la précision et 35% pour le rappel. Ce qui est tout juste supérieur à ce que nous avons obtenu pour le moteur précédent.

Deux tableaux récapitulatifs, le 5.4 et le 5.5 reprennent respectivement les différentes valeurs de l'indicateur de précision et de l'indicateur de rappel pour chaque requête et chaque nombre de documents.

TABLE 5.4 – Valeurs en pourcent de l'indicateur « précision » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur avec ontologie

Requête	1	2	3	4	5	6	7	8	9	10
Vador	100%	100%	67%	75%	80%	83%	71%	75%	78%	70%
Naboo	0%	50%	33%	50%	40%	50%	57%	63%	67%	60%
Palpatine	0%	0%	33%	25%	40%	50%	57%	63%	56%	60%
Sidious	0%	0%	33%	25%	40%	50%	43%	38%	44%	50%
csi	100%	50%	33%	50%	60%	50%	43%	38%	44%	40%
Moyenne	40%	40%	40%	45%	52%	57%	54%	55%	58%	56%

TABLE 5.5 – Valeurs en pourcent de l'indicateur « rappel » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur avec ontologie

Requête	1	2	3	4	5	6	7	8	9	10
Vador	6%	12%	12%	18%	24%	29%	29%	35%	41%	41%
Naboo	0%	8%	8%	15%	15%	23%	31%	38%	46%	46%
Palpatine	0%	0%	6%	6%	11%	17%	22%	28%	28%	33%
Sidious	0%	0%	6%	6%	11%	17%	17%	17%	22%	28%
csi	7%	7%	7%	13%	20%	20%	20%	20%	27%	27%
Moyenne	3%	5%	7%	11%	16%	21%	24%	28%	33%	35%

5.5 Moteurs de recherche documentaire existants

La suite de l'évaluation a pour but de comparer ce qui a été réalisé avec des moteurs de recherche existants. Nous en avons retenu deux : le moteur de recherche interne de « `Starwars-Holonet.com` » et le moteur de recherche `Google` restreint au site « `Starwars-Holonet.com` ».

La première requête, « Vador » retourne cinq résultats positifs pour `Holonet` et cinq également pour `Google`. Ce qui nous donne une précision de 50% et un rappel de 29% dans les deux cas.

La deuxième requête concerne « Naboo ». `Holonet` retourne huit résultats positifs, ce qui donne une précision de 80% et un rappel de 62%. `Google`, lui, ne retourne que six résultats pertinents ce qui donne 60% pour la précision et 46% pour le rappel.

La troisième requête, « Palpatine », retourne cinq résultats pour `Holonet` et sept pour `Google`. On a donc une précision et un rappel de respectivement 50% et 28% pour le premier et 70% et 39% pour le second.

La quatrième requête est celle sur « Sidious ». Nous obtenons quatre résultats pertinents pour `Holonet` et pour `Google` ce qui donne donc une précision de 40% et un rappel de 22% pour les deux.

TABLE 5.6 – Valeurs en pourcent de l'indicateur « précision » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur `Holonet`

Requête	1	2	3	4	5	6	7	8	9	10
Vador	100%	50%	67%	75%	60%	67%	57%	63%	56%	50%
Naboo	100%	100%	67%	50%	60%	67%	71%	75%	78%	80%
Palpatine	100%	100%	67%	50%	60%	50%	43%	50%	56%	50%
Sidious	100%	50%	67%	50%	40%	50%	57%	50%	44%	40%
csi	0%	0%	33%	50%	60%	67%	57%	50%	56%	60%
Moyenne	80%	60%	60%	55%	56%	60%	57%	58%	58%	56%

TABLE 5.7 – Valeurs en pourcent de l'indicateur « rappel » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur `Holonet`

Requête	1	2	3	4	5	6	7	8	9	10
Vador	6%	6%	12%	18%	18%	24%	24%	29%	29%	29%
Naboo	8%	15%	15%	15%	23%	31%	38%	46%	54%	62%
Palpatine	6%	11%	11%	11%	17%	17%	17%	22%	28%	28%
Sidious	6%	6%	11%	11%	11%	17%	22%	22%	22%	22%
csi	0%	0%	7%	13%	20%	27%	27%	27%	33%	40%
Moyenne	5%	8%	11%	14%	18%	23%	26%	29%	33%	36%

TABLE 5.8 – Valeurs en pourcent de l'indicateur « précision » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur `Google`

Requête	1	2	3	4	5	6	7	8	9	10
Vador	100%	100%	67%	75%	60%	50%	57%	63%	56%	50%
Naboo	100%	100%	100%	75%	60%	50%	43%	50%	56%	60%
Palpatine	100%	100%	100%	75%	80%	83%	71%	63%	67%	70%
Sidious	100%	50%	67%	50%	40%	33%	43%	38%	44%	40%
csi	100%	100%	100%	75%	80%	67%	57%	63%	67%	70%
Moyenne	100%	90%	87%	70%	64%	57%	54%	55%	58%	58%

La cinquième requête, « csi » donne six et sept résultats respectivement pour `Holonet` et `Google`. On obtient donc une précision et un rappel de 60% et 40 pour le premier et de 70% et 47 pour le second.

En faisant la moyenne sur l'ensemble des requêtes, `Holonet` obtient 56% pour la pertinence et 36% pour le rappel, ce qui est juste 1% de plus que le moteur de recherche avec l'*ontologie*.

Pour `Google`, la précision moyenne est de 58% et le rappel est à 37% ce qui le place en première position.

Quatre tableaux récapitulatifs, le 5.6, le 5.7, le 5.8 et le 5.9 reprennent respectivement les différentes valeurs de l'indicateur de précision et de l'indicateur de rappel pour chaque requête et chaque nombre de documents pour `Holonet` suivi des mêmes tableaux pour `Google`.

TABLE 5.9 – Valeurs en pourcent de l'indicateur « rappel » en fonction du nombre de résultats pour les différentes requêtes exécutées par le moteur `Google`

Requête	1	2	3	4	5	6	7	8	9	10
Vador	6%	12%	12%	18%	18%	18%	24%	29%	29%	29%
Naboo	8%	15%	23%	23%	23%	23%	23%	31%	38%	46%
Palpatine	6%	11%	17%	17%	22%	28%	28%	28%	33%	39%
Sidious	6%	6%	11%	11%	11%	11%	17%	17%	22%	22%
csi	7%	13%	20%	20%	27%	27%	27%	33%	40%	47%
Moyenne	6%	11%	17%	18%	20%	21%	24%	28%	33%	37%

5.6 Analyse des résultats obtenus

On a donc des métriques pour cinq requêtes et quatre moteurs de recherche documentaire. Si on prend la moyenne des métriques pour l'ensemble des requêtes, `Google` sort en tête, suivi de près par `Holonet` et immédiatement en dessous notre moteur de recherche avec l'*ontologie* et enfin, juste en dessous, notre moteur de recherche sans l'*ontologie* qui finit dernier.

En regardant de plus près, on constate que notre moteur de recherche a un très mauvais résultat pour la dernière requête et ce que ce soit avec ou sans utiliser l'*ontologie*. Si on élimine cette requête, notre moteur de recherche passe devant les deux moteurs existants avec une avance de 5% sur la précision et d'un peu plus de 2% pour le rappel.

Il semblerait donc que l'échantillon de requêtes soit trop petit pour tirer une réponse claire. On constate que pour certaines requêtes notre système apporte un plus et que pour d'autre, il est à la traîne.

Revenons à la dernière requête qui fait chuter les indicateurs de notre moteur de recherche documentaire et tentons d'expliquer à quoi cela est dû. Si on regarde cette requête en tant qu'expert du domaine de `Star Wars`, on sait que le terme « csi » de même que le terme « séparatiste » qui fait référence à la même organisation est un terme très à la mode et très central dans l'univers de `Star Wars` actuel. L'organisation à laquelle ces deux termes font référence est un acteur central de toute une partie de la saga et elle a changé radicalement la tournure des choses. Pratiquement chaque évènement, chaque personne et chaque organisation de cette époque a été influencé d'une manière ou d'une autre par l'organisation appelée le « csi ». Á partir de là, on trouve des références vers elle un peu partout. Que ce soit dans les différentes organisations plus petites qui la composent, dans les personnages qui font partie de cette organisation ou qui ont tout perdu à cause d'elle, dans les vaisseaux construits par cette organisation ou pour lutter contre elle, ... Devant cette relation présente entre cette organisation et pratiquement tout ce qui a eu lieu pendant sa période d'activité ou juste après, nous pourrions donc en conclure que les termes sont devenus trop génériques pour que notre moteur de recherche puisse faire la différence entre les fiches qui parlent de choses qui sont vaguement reliées à cette organisation et les fiches qui sont vraiment pertinentes pour cette même organisation.

Chapitre 6

Conclusion

Le but de ce mémoire était de mettre an point un moteur de recherche avec une distance sémantique plus faible entre la question de l'utilisateur et la requête qui permettrait au système d'y répondre et de voir si cela apportait une amélioration dans l'utilisation du système. Le choix qui a été fait est d'utiliser une *ontologie* pour détecter des termes différents faisant référence à une même instance et donc à une même « chose » dans la tête de l'utilisateur.

Nous avons commencé par faire un tour d'horizon de ce qu'il faut savoir sur l'*indexation*, sur les moteurs de recherche documentaire et sur les *ontologies*. Ensuite, nous avons choisi les méthodologies à appliquer pour développer et évaluer un moteur de recherche documentaire utilisant une *ontologie*. Après ça, nous avons construit un indexeur et une *ontologie* et nous avons cherché un moyen de les faire travailler ensemble. À partir de ça, nous avons utilisé ce que nous venions d'élaborer pour obtenir notre propre moteur de recherche documentaire. Et enfin, nous avons effectué une petite étude de cas pour essayer de voir si le système que nous venions tout juste de développer apporte quelque chose de plus, si notre système permet d'obtenir de meilleurs résultats pour une même requête qu'un système plus classique.

Nous avons réussi à faire travailler ensemble un indexeur de documents et une *ontologie*. Nous avons basé notre moteur de recherche sur cela. Nous nous sommes contenté d'une toute petite *ontologie* et d'une interaction très simple entre l'indexeur et celle-ci mais l'interaction est là. Si ce travail était à refaire, un intérêt plus grand serait porté à l'interaction pour obtenir quelque chose tirant un meilleur profit de l'*ontologie* et donc montrer plus facilement le gain que pourrait apporter un système comme celui que nous avons développé.

La petite étude de cas réalisée au chapitre 5 ne nous permet pas de montrer que notre système est meilleur ou plus mauvais qu'un système plus classique. On constate que d'une requête à l'autre la balance penche plus d'un côté ou de l'autre. Si bien qu'il semble nécessaire de faire des tests à plus grande échelle en augmentant le nombre de résultats (qui est ici fixé à dix) et le nombre de requêtes (qui est ici de cinq) pour tenter d'apporter une réponse plus claire à cette question importante.

Mais si nous regardons de plus près les résultats de l'étude de cas et que nous les rapprochons au fait qu'on a qu'une toute petite interaction avec l'*ontologie*, il semblerait que cette piste mériterait d'être poursuivie. Un exemple d'amélioration de notre système pourrait passer par l'exploitation de relations de l'*ontologie* pour ne plus se contenter d'identifier les instances ayant des termes pertinents pour une requête donnée mais bien d'essayer de trouver en plus de ces instances là des relations qui partent d'elles et mènent à d'autres instances elles aussi intéressantes pour la requête.

Bibliographie

[jOp, 2008] (2008). *jOpenDocument - A pure Java library for OASIS Open Document files manipulation.* url : http ://www.jopendocument.org – consulté le 17 mars 2009. 54

[luc, 2009] (2009). *Apache Lucene - Le site web.* http ://lucene.apache.org – consulté le 17 mars 2009. 28

[jen, 2009] (2009). *La documentation de Jena.* http ://-jena.sourceforge.net/documentation.html – consulté le 17 mars 2009. 66

[pro, 2009] (2009). Le site web du logiciel protégé. http ://protege.stanford.edu – consulté le 17 mars 2009. 57

[Bachimont, 2000] Bachimont, B. (2000). Engagement sémantique et engagement ontologique : conception et réalisation d'ontologies en ingénierie des connaissances. *Ingénierie des connaissances : évolutions récentes et nouveaux défis*, pages 305–324. 33, 40

[Cao, 2004] Cao, G. (2004). Présentation des généralités de la RI. 9, 20, 21, 25, 34, 35, 36, 37, 38

[Cimiano, 2006] Cimiano, P. (2006). *Ontology Learning and Population from Text.* Springer. ISBN : 0-387-30632-3. 29, 93

[Gruber, 1995] Gruber, T. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human Computer Studies*, 43(5):907–928. 29

[Hainaut, 2005] Hainaut, J.-L. (2004-2005). Ingénierie de base de données. Syllabus du cours de 3ème année de Bachelier en Informatique aux FUNDP. 9, 19, 20, 21

[Hatcher et Gospodnetić, 2004] Hatcher, E. et Gospodnetić, O. (2004). *Lucene in Action.* Manning. ISBN : 1932394281. 28

[Horrocks, 2007] Horrocks, I. (2007). Semantic web : the story so far. *In W4A '07 : Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 120–125, New York, NY, USA. ACM. 34

[Horrocks *et al.*, 2003] Horrocks, I., Patel-Schneider, P. F. et van Harmelen, F. (2003). From $\mathcal{SHIQ}$ and RDF to OWL : The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26. 34

[Konchady, 2006] Konchady, M. (2006). *Text Mining Application Programming*. Charles River Media. 93

[Macklovitch, 2008] Macklovitch, E. (2008). Les travaux du laboratoire rali en traitement automatique de la langue (tal). RALI. Conférence organisées aux FUNDP. 95

[Noy *et al.*, 2005] Noy, N. F., McGuinness, D. L. et Angjeli, A. (2005). *Développement d'une ontologie 101 : Guide pour la création de votre première ontologie*. Université de Stanford, Stanford, CA, 94305. 28, 29, 31, 39, 40, 57

[Oliver *et al.*, 2008] Oliver, A. C., Stampoultzis, G., Sengupta, A. et Klute, R. (2008). Apache poi - java api to access microsoft format files. site web. url : http ://-poi.apache.org – consulté le 17 mars 2009. 53

[Prud'Hommeaux *et al.*, 2006] Prud'Hommeaux, E., Seaborne, A. *et al.* (2006). SPARQL query language for RDF. *W3C working draft*, 4. 67

[Spyns *et al.*, 2002] Spyns, P., Meersman, R. et Jarrar, M. (2002). Data modelling versus ontology engineering. *ACM SIGMOD Record*, 31(4):12–17. 30

Annexe A

Term frequency – Inverse document frequency

La notion de *TF-IDF* est une méthode de pondération de la présence d'un terme dans un document particulier en tenant compte des autres termes de ce document et du nombre de documents dans lequel le terme apparaît.

A.1 Term frequency

Cette notion représente l'importance d'un terme dans un document particulier indépendamment des autres documents.

Pour un terme i et un document d_j :

$$tf_{i,j} = \frac{n_{i,j}}{\sum_{k=1}^{l_j} n_{k,j}} \tag{A.1}$$

où $n_{a,b}$ représente le nombre d'occurrences du terme a dans le document d_b,
l_j représente le nombre de termes différents présents dans le document d_j.

A.2 Inverse document frequency

Cette notion représente l'importance générale d'un terme dans l'ensemble des documents.

Pour un terme i :

$$idf_i = log\frac{|D|}{|\{d_j : t_i \in d_j\}|} \tag{A.2}$$

où $|D|$ représente le nombre total de documents,
$|\{d_j : t_i \in d_j\}|$ représente le nombre de documents où apparaît le terme t_i (autrement dit tel que $n_ij \neq 0$)

A.3 Term frequency – Inverse document frequency

En multipliant les 2 notions précédentes entre elles, on obtient un indicateur qui représente l'importance d'un terme dans un document particulier tout en tenant compte des autres

documents. Une valeur élevée de cet indicateur indique que le terme est fréquent dans le document tout en étant présent dans peu d'autres documents.

$$tfidf_{i,j} = tf_{i,j} * idf_i \tag{A.3}$$

Annexe B

Captures d'écran de l'exécution de l'indexeur

Cette annexe regroupe l'ensemble des captures d'écran de l'interface graphique de l'indexeur de document réalisé dans le cadre de ce travail. Les commentaires associés aux différentes figures sont les mêmes que ceux déjà présents dans le texte décrivant l'indexeur à la section 4.2.

L'image B.1 est une capture d'écran du processus de collecte de l'indexeur. On peut voir sur cette image trois graphiques. Le premier, en haut à gauche, représente la répartition des types de documents en pourcentage du nombre total de documents. Le deuxième graphique, en haut à droite, montre le premier tri des documents, il va montrer le pourcentage de documents collectés qui seront refusés à ce niveau et ne seront donc pas traités. Enfin, le troisième et dernier graphique, celui du bas, montre pour chaque collecteur actif le nombre de documents collectés. Ces trois graphiques sont mis à jour en temps réel pendant l'exécution de cette phase par l'indexeur.

L'image B.2 est une capture d'écran du processus de *parsing* de l'indexeur. On peut y voir trois graphiques. Le premier, en haut à gauche, représente la répartition des documents entre les différents *parsers*. Le deuxième, en haut à droite, montre la répartition des documents en fonction de leur statut (en attente, traité, pas traité pour cause d'erreur, ...). Enfin, le troisième et dernier graphique, en bas, représente le nombre de documents traités en fonction du temps que cela a pris et ce par *parser*.

L'image B.3 est une capture d'écran du processus d'écriture de l'*index* dans l'indexeur. L'écriture de *index* par la librairie *Lucene* se résume vraiment à la toute dernière étape et se fait dans un temps inférieur à la milliseconde pratiquement dans tous les cas ce qui rend ce graphique bien moins intéressant que ceux des phases précédentes.

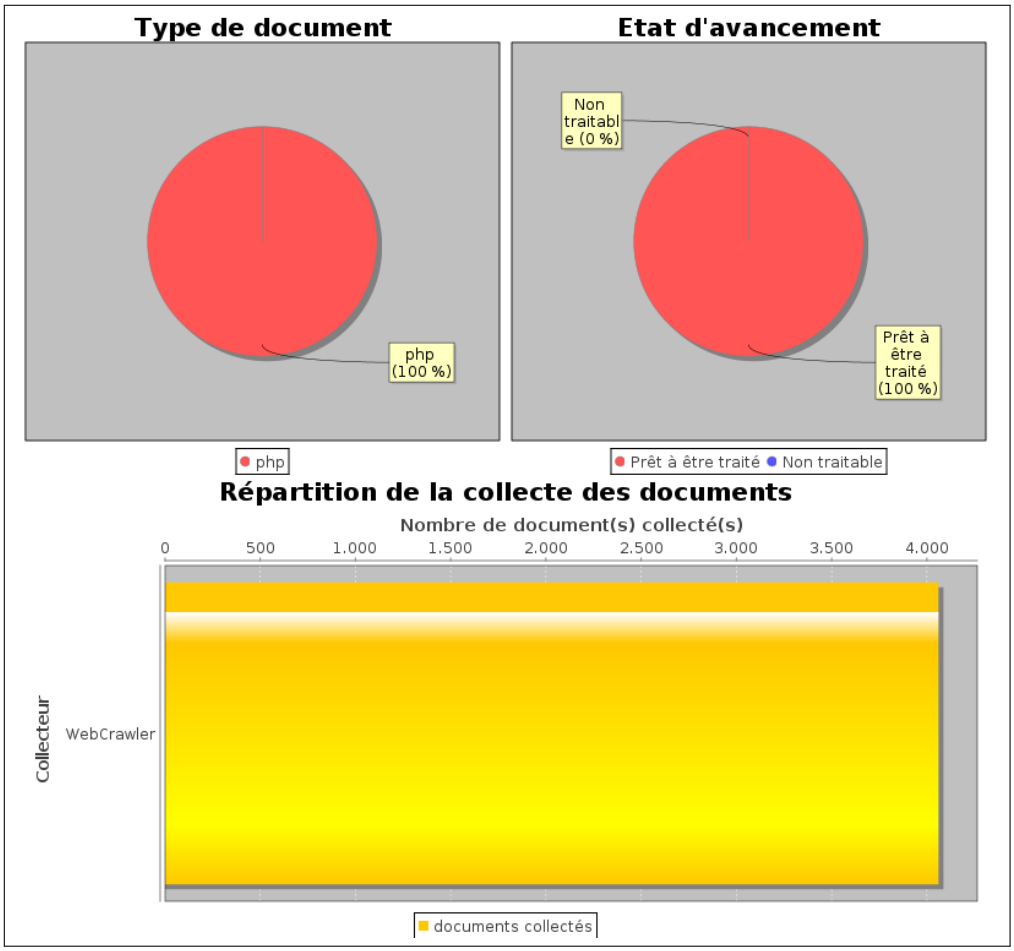

FIGURE B.1 – Capture d'écran du processus de collecte

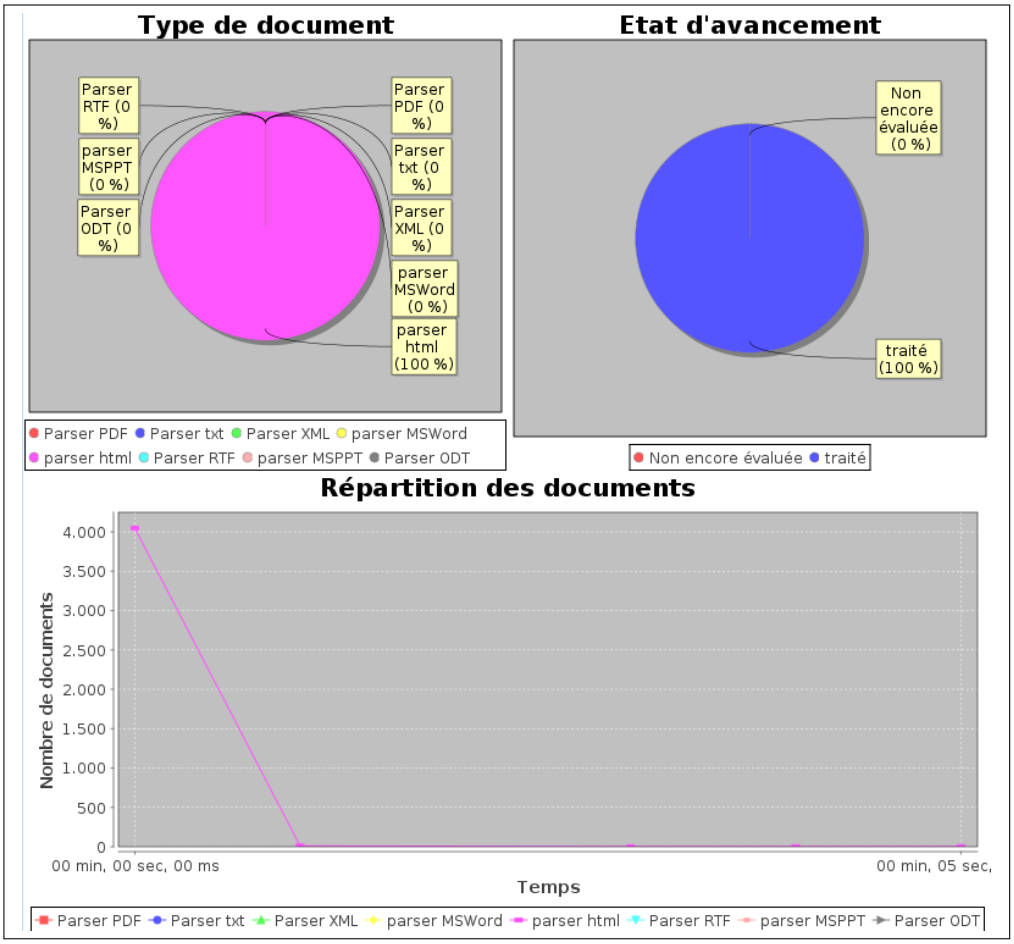

FIGURE B.2 – Capture d'écran du processus de parsing

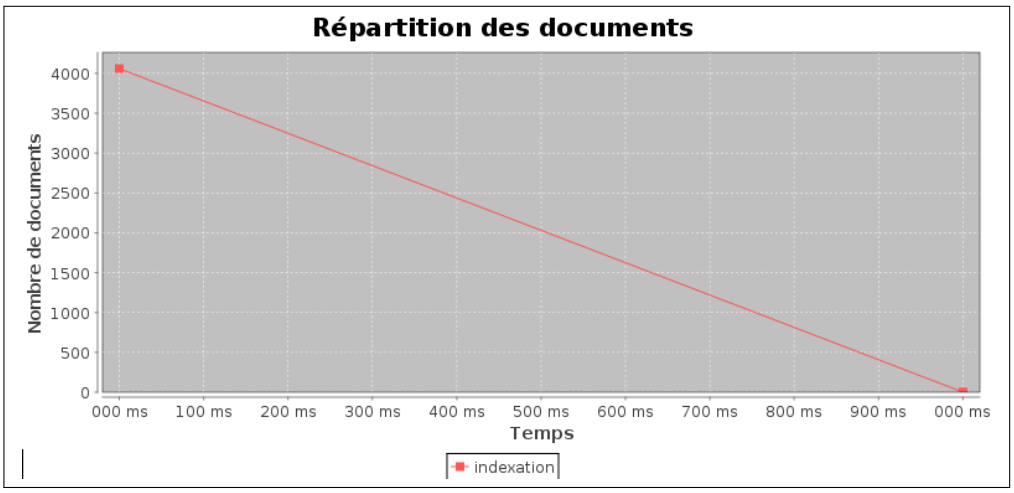

FIGURE B.3 – Capture d'écran du processus d'écriture de l'index

Annexe C

Rapport de stage

C.1 Introduction

Mon stage s'est déroulé au *CETIC* à *Gosselies* dans l'équipe *Traitement sémantique de l'information (TSI)* sous la supervision de Fabrice `Estiévenart` et avec l'aide de Joseph `Roumier`. Le premier contact que j'ai eu pour mon stage a consisté en une réunion début juillet au *CETIC* avec mon promoteur, Fabrice `Estiévenart` et Joseph `Roumier`. Nous y avons discuté de l'état de mes connaissances, de ce qui serait le corps de mon stage et des dates de début et fin effectives de mon stage. Celui-ci devait débuter réellement le 1^{er} septembre et se terminer le 31 décembre avec possibilité de déborder jusqu'au 15 janvier si nécessaire. En plus de cela, nous avons convenu d'une série de lectures intéressantes dans le cadre du stage dont deux livres : [Cimiano, 2006, Konchady, 2006].

Le contenu du présent document se base très fortement sur la description des objectifs de mon stage rédigé par Fabrice `Estiévenart` dès le début de celui-ci.

C.2 Objectifs du stage

Avant même le début du stage, 3 objectifs ont été clairement identifiés :

1. Comprendre et étudier comment un ensemble de connaissances (modélisé sous la forme d'une ontologie) peut améliorer l'efficacité d'un processus d'indexation et de recherche de documents.

2. Développer un système d'indexation de documents basé sur des critères sémantiques.

3. Mettre en pratique le système dans le cadre d'une étude de cas.

Une fois le stage commencé, un certain nombre d'autres objectifs sont venus se rajouter à la liste :

4. Le *CETIC* voulait mettre en avant l'Open Source et j'étais donc invité à utiliser de telles composantes aussi souvent que possible.

5. Mon principal objectif personnel était d'arriver à un **prototype** permettant de montrer les éventuels bénéfices d'un indexeur sémantique en faisant abstraction d'un certain nombre de considérations et cela à donc guider un certain nombre de choix.

6. Le système à construire devait être capable :

   (a) de collecter des documents électroniques sur un système de fichiers (unité = le fichier), sur un ensemble de sites web (unité = la page), dans une boîte aux lettres électronique (unité = l'émail et ses pièces jointes), à partir d'un flux *RSS*.

   (b) d'extraire le contenu textuel des types de documents les plus fréquents.

   (c) d'analyser le contenu des documents afin d'en extraire les termes à indexer.

   (d) d'indexer les termes selon un format permettant une recherche rapide et efficace (probablement le format *Lucene*.

   (e) de fournir un rapport sur le déroulement du processus d'indexation.

   (f) de proposer un module de recherche « intelligent ». Parmi les fonctionnalités avancées, on peut citer :

      i. la suggestion de documents similaires, du point de vue syntaxique
      ii. la suggestion de documents similaires, du point de vue sémantique, i.e. traitant du même sujet
      iii. la suggestion de documents traitant de sujets connexes, i.e. proches sur le plan sémantique
      iv. le regroupement selon différents critères : auteur, date de modification, répertoire de stockage...

C.3 Évolutions des objectifs au cours du stage

Un autre objectif est venu se rajouter à la liste dans le courant du stage. En effet, le *CETIC* a détecté un intérêt possible aussi bien en interne qu'en externe pour un moteur de recherche documentaire personnalisé. Il en résultait donc une utilisation potentielle de mon travail pour fournir un système opérationnel ce qui entrait clairement en opposition avec l'objectif 5 de la section précédente. Cela a renforcé le besoin de documentation de chaque choix effectué dans le cadre du développement pour pouvoir par la suite revenir sur ceux-ci et pouvoir les modifier pour produire le système opérationnel voulu.

L'élaboration d'une ontologie prenant plus de temps que prévu, nous avons dû revoir les objectifs pour la fin. L'étude de cas a été complètement annulée dans le cadre du stage et prendra plutôt place dans le cadre du mémoire.

Mon chef de stage avait manifesté un grand intérêt pour le traitement des boîtes aux lettres électroniques (objectif 6a) mais le traitement de tels documents n'apportait pas vraiment de plus dans le cadre d'un indexeur sémantique et nous avons donc décidé de

ne pas considéré ce point précis comme prioritaire et de le traiter à la fin s'il restait du temps ce qui n'a pas été le cas.

C.4 Déroulement du stage

La première chose à dire en ce qui concerne le déroulement du stage est, selon moi, de préciser que j'ai envoyé chaque semaine un émail succinct à mon promoteur avec un résumé de ce que j'avais fait la semaine précédente pour l'en tenir informé.

L'accueil sur mon lieu de stage s'est très bien passé. En effet, le jour de mon arrivée, tout était prêt. Un bureau et une machine était directement à ma disposition et le choix parmi 2 systèmes d'exploitations : *Windows* et *Linux*. Le tout déjà configuré si bien que je pouvais directement me mettre au travail. En plus de ça, le *CETIC* a mis à ma disposition un *Système de gestion de versions (SVN)* et un *Bug Tracker* en m'encourageant à les utiliser pour rendre mon travail plus propre. A ce propos, la configuration des droits d'accès du *Bug Tracker* a posé quelques problèmes car je ne pouvais qu'ajouter de nouvelles tâches ou de nouveaux bugs. Pour toute autre utilisation de l'outil, je devais passer par mon supérieur, Fabrice `Estiévenart` pour que lui le fasse ce qui rendait les choses plus contraignantes et gâchait donc un peu les bénéfices de l'outil.

Un autre détail important à signaler sur le déroulement de mon stage est que j'étais convié aux réunions de communications du centre et également à la journée de *Team Building* où j'ai pu avoir une meilleure vision du *CETIC* dans son ensemble et me sentir très bien intégré dans le centre.

En plus des réunions de communications, le *CETIC* organise fréquemment ce qu'ils appellent des *Technical Information Meeting (TIM)* où un membre du personnel ou un stagiaire explique son travail au reste du centre d'une manière qui se veut accessible et interactive. Cela permet d'échanger des connaissances de manière agréable. J'ai d'ailleurs présenté moi-même un *TIM* pour expliquer les résultats de mon stage à l'ensemble du centre.

Un autre détail utile à préciser concernant le déroulement du stage est qu'une proposition d'échéancier avait été rédigé par Fabrice `Estiévenart` mais ne correspondait pas du tout à mes habitudes de travail et nous ne l'avons donc pas suivi. Quand nous avons revus les objectifs vers la fin du stage au vu du retard que l'on avait, on a également défini un nouvel échéancier pour le temps restant pour arriver à un stade satisfaisant pour tout le monde.

C.5 Activité complémentaire

En plus d'avoir travaillé au *CETIC* j'ai assisté à une conférence [Macklovitch, 2008] organisée aux *FUNDP* le jeudi 27 novembre ayant pour thème le traitement automatique

de la langue et en particulier l'extraction d'information, le résumé automatique et la traduction automatique. L'orateur de la conférence était M. Elliott `Macklovitch`, coordinateur du *Laboratoire de Recherche Appliquée en Linguistique Informatique (RALI)* de l'Université de *Montréal*. Le sujet n'entrait pas directement en rapport avec ce qui allait être développé dans le cadre de mon stage mais pouvait cependant être intéressant pour mon futur mémoire et intéressait également mon chef de stage, Fabrice `Estiévenart`, ce qui nous a poussé à nous rendre à cette conférence ensemble.